VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Kompresní a obfuskační metody užívané u malware: detekce a automatická dekomprimace

Packers and Methods of Obfuscation Used by Malware: Detection and Automatic Decompression

2019 Bc. Jaroslav Seidel

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jaroslav Seidel**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 1801T064 Informační a komunikační bezpečnost

Téma: Kompresní a obfuskační metody užívané u malware: detekce a automatická dekomprimace
Packers and Methods of Obfuscation Used by Malware: Detection and Automatic Decompression

Jazyk vypracování: čeština

Zásady pro vypracování:

Autoři malware často užívají kompresní a obfuskační metody k zamaskování těla škodlivých programů. Student provede rešerši metod, které se snaží detekovat, zda je program chráněn některými z maskovacích metod. Dále se student zaměří na konkrétní známé, i méně známé, kompresní a obfuskačních metody - ty popíše a provede srovnání. Dále se pokusí nalézt existující algoritmy a nástroje sloužící k jejich detekci, případně i dekomprimaci.
V praktické části využije student získané informace k vytvoření analyzační aplikace, která v první fázi zjistí, zda je tělo zkoumaného souboru komprimováno nebo jinak chráněno. Následně se pokusí detekovat užitou metodu komprese či obfuskace, v poslední fázi se pokusí o dekomprimaci. Aplikace bude pracovat se spustitelnými "exe" soubory a bude umět zpracovávat soubory dávkově a plně automaticky.

Hlavní body zadání:
1. Rešerše metod sloužících k detekci komprimovaných či obfuskovaných spustitelných souborů.
2. Srovnání nalezených metod, algoritmů a nástrojů.
3. Vývoj nástroje k automatické detekci a dekomprimaci chráněných souborů.
4. Testování a srovnání výsledků s již existujícími nástroji.

Seznam doporučené odborné literatury:

[1] SIKORSKI, Michael a Andrew HONIG. Practical malware analysis: the hands-on guide to dissecting malicious software. San Francisco: No Starch Press, 2012. ISBN 978-1-59327-290-6.
[2] DANG, Bruce., Alexandre. GAZET, Elias. BACHAALANY a Sébastien. JOSSE. Practical reverse engineering: x86, x64, ARM, Windows Kernel, reversing tools, and obfuscation. Indianapolis, IN: John Wiley, 2014. ISBN 9781118787250.
[3] Malware analyst's cookbook and DVD: tools and techniques for fighting malicious code. Indianapolis: Wiley, c2011. ISBN 978-0-470-61303-0.
[4] EILAM, Eldad. Reversing: Secrets of reverse engineering. Indianapolis: Wiley, c2005. ISBN 0-7645-7481-7.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **prof. Ing. Ivan Zelinka, Ph.D.**

Datum zadání: 01.09.2018

Datum odevzdání: 30.04.2019

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. dubna 2019 Seidel .....

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.

V Ostravě 30. dubna 2019 ..... Seidel .....

Rád bych na tomto místě poděkoval všem, kteří mě během studia podporovali. Dále děkuji mému vedoucímu diplomové práce prof. Ing. Ivanovi Zelinkovi Ph.D. za vedení této práce.

Abstrakt

Tato diplomová práce se zabývá kompresními a obfuskačními technikami používaných k ochraně malwaru i jiného softwaru. Jsou představeny nástroje pro kompresi spustitelných PE souborů zvané jako packery, metody pro jejich detekci a dekompresi. V praktické části probíhá tvorba analyzační aplikace k detekci komprimovaných souborů a použitého packeru. V práci je dále provedena řada experimentů s komprimovanými soubory a antivirovými programy.

Klíčová slova: Komprese, obfuskace, šifrování, dekomprese, packery, malware

Abstract

This thesis deals with compression and obfuscation techniques used to protect malware and other software. PE file compression tools called packers, methods for their detection and decompression are introduced. In the practical part there is created an analysis application for detection of packed files and used packer on them. Furthermore, a series of experiments with packed files and antivirus programs are performed.

Key Words: Compression, obfuscation, encryption, decompression, packers, malware

Obsah

Seznam použitých zkratek a symbolů

API	–	Application Programming Interface
ASCII	–	American Standard Code for Information Interchange
AV	–	Anti-virový
CPU	–	Central Processing Unit
DLL	–	Dynamic-Link Library
DRM	–	Digital Rights Management
ELF	–	Executable and Linkable Format
EP	–	Entry Point
EXE	–	Executable (formát spustitelného souboru, přípona souboru)
FP	–	False-Positive
FUU	–	Faster Universal Unpacker
GNU	–	GNU's Not Unix!
GUI	–	Graphical User Interface
MACH-O	–	Mach Object File Format
MIME	–	Multipurpose Internet Mail Extensions
MZ	–	Mark Zbikowski
OEP	–	Original Entry Point
OS	–	Operační Systém
PE	–	Portable Executable
ROT13	–	Rotate by 13 places (název šifry)
UPX	–	Ultimate Packer for eXecutables
URL	–	Uniform Resource Locator
XOR	–	Exkluzivní disjunkce (exkluzivní OR)

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

V dnešním kyberprostoru existuje spousta malwaru. Jejich autoři se již od počátku vzniku prvních škodlivých programů zabývají problémem, jak zabránit jejich detekci a zpětné analýze. Jejich největším nepřítelem jsou v současnosti antivirové programy, které využívají čím dál více sofistikovanější metody detekce. Aby takový malware mohl co nejdéle přežít, jejich autoři často využívají implementací ochranných metod. Nejčastějšími z nich jsou obfuskace, komprese a šifrování. Tyto metody nejsou však používány pouze u malwaru, ale i u ostatních softwarů, jejichž autoři chtějí zabránit neoprávněnému šíření svých know-how nebo vzniku pirátských kopií.

V teoretické části se práce nejprve zabývá kompresními, obfuskačními a šifrovacími metodami malwaru. Je zmíněna historie maskovaní jejich těl a následně situace v současné době. Je popsána struktura formátu spustitelných souborů PE na platformě Windows, kterou je důležité znát při statické i dynamické analýze malwaru. Velká část je věnována kompresi spustitelných souborů. Jsou představeny nástroje zvané packery, které nabízí různé funkcionality k ochraně spustitelných souborů včetně kompresne, šifrování i obfuskace. Další část se věnuje metodám detekce komprimovaných souborů a packeru, který byl na ně použit. Jsou také představeny nástroje, které tyto metody detekce využívají. Nezapomenuta je také dekomprese, která může být manuální nebo automatická, a různé nástroje pro automatickou dekompresi. Poslední kapitoly teoretické části se zabývají obfuskačními metodami na zdrojovém kódu assembleru nebo vyšších programovacích jazyků.

V praktické části je popsána tvorba analyzační aplikace, která se snaží detekovat, zda je zkoumaný soubor komprimovaný a jakým packerem. Jsou implementovány dvě metody detekce komprimovaného souboru, které jsou inspirovány již existujícími studiemi. První metoda detekuje komprimovaný soubor podle číselných hodnot různých vlasností PE souboru, které jsou předem zjištěny či spočítány. Druhá metoda se jej snaží detekovat pomocí klasifikačních algoritmů strojového učení, které jsou natrénovány předem vytvořeným datovým setem. Algoritmů strojového učení je také využito k detekci použitého packeru na komprimovaný soubor. V další části je provedena dekomprese automatická, pomocí nástrojů představených v teoretické části, a manuální, pomocí disassembleru a debuggeru OllyDbg. U obou způsobů je rozebrána struktura PE souboru před a po dekompresi. Poslední část je věnována různým experimentům a pozorováním, jako např. sledování entropie komprimovaných souborů, experiment s antivirovými programy, nebo experiment s vícenásobnou kompresí.

2 Teoretická část

2.1 Účel komprese, šifrování a obfuskace

Komprese, šifrování a obfuskace je způsob transformace kódu softwaru, která jej způsobuje tíže pochopitelným člověkem nebo programem (např. disassemblerem), aniž by byla narušena jeho původní funkčnost. Důvody, proč při tvorbě softwaru použít tyto metody jsou nejčasteji:

- **Malware** - Obfuskací se malware stává hůře detekovatelný antivirovými programy a reverzním inženýrům je znesnadněna zpětná analýza malwaru.
- **Ochrana duševního vlastnictví** - Některé softwarové společnosti chtějí skrýt určitou implementaci svého programu proti neoprávněnému šíření. Může se například jednat o různé algoritmy nebo protokoly. Příkladem je komunikační protokol Skypu, který obsahuje vrstvu pro obfuskování datového obsahu v datagramech.
- **Digital rights management** - metody DRM (v češtině správy digitálních práv) často využívají obfuskaci k ochraně digitálních médií (filmy, hudba, elektronické knihy, atd.) za účelem užívání jejich obsahu v souladu s licenčními podmínkami a autorskými právy. V takovém případě může být obfuskace použita například jako otisk do softwaru, který bude mít každá vydaná licence unikátní. Pokud tak budou nalezeny nelegální kopie softwaru, bude snadné dohledat osobu, která je distribuovala.

Transformaci softwaru kompresí, šifrováním nebo obfuskací lze rozdělit na:

- **Transformace binárních souborů** - program je již zkompilován do spustitelného souboru PE formátu (EXE, DLL), na který je použit packer
- **Transformace zdrojového kódu** - obfuskace zdrojového kódu konkrétních programovacích jazyků (Assembler, C, C++, Java, C#, apod.) dříve, než je zkompilován do binární podoby

2.2 Výhody a nevýhody

Komprese, šifrování a obfuskace kódu softwaru s sebou nesou určité výhody a nevýhody. Přináší nám ochranu programu před zpětnou analýzou, nicméně to nás může stát výpočetní čas. Hlavní výhody a nevýhody těchto transformací jsou[37]:

Výhody:

- Ochrana - Brání proti statické a dynamické analýze kódu a reverzní inženýr tak musí nejpre provést dekompresi nebo deobfuskaci.

- Různorodost - Je možné vytvořit několik odlišných instancí téhož programu (u virů například polymorfní a metamorfní viry).
- Udržitelnost - Díky automatickému procesu dekomprese nebo deobfuskace a kompatibilitě s existujícími systémy nevyžaduje vysokou údržbu.
- Nezávislost na platformě - Obfuskace zdrojového kódu může být aplikována na vyšších programovacích jazycích, takže nemusí být nutně závislá na použitém operačním systému či sadě instrukcí.

Nevýhody:

- Nároky na výkon - Komprimovaný, šifrovaný či obfuskovaný program může spotřebovávat více paměti a výpočetního času počítače, zejména kvůli potřebnému procesu dekomprese a deobfuskace.
- Prolomitelná ochrana - Komprese, šifrování nebo obfuskace nepřináší dokonalou ochranu proti zpětné analýze, pouze ji dělá obtížnější, ne však nemožnou.

2.3 Historie maskování malwaru

Již v 80. letech 20. století si autoři prvních malwarů pro MS-DOS uvědomovali, že pokud mají jejich výtvory nadále vzrůstat a rozšiřovat se, musí jim zajistit přežití v kybersvěte. Kvalita utajení malwaru v systému a vysoká odolnost proti zpětné analýze jen zvyšovaly šanci, že se bude virus nadále šířit a vyvíjet.

Prvním malwarem, který se snažil v systému zamaskovat byl virus Brain. Byl vyvinut v roce 1986 bratry Farooq Alvi z Pákistánu a je obecně znám jako vůbec první virus pro MS-DOS. Infikoval počítač přehráním původního boot sektoru infikovaným z diskety. Původní boot sektor se přesunul do jiné části disku a byl označen jako poškozený. Označení disku se například z `C:\` změnilo na `@Brain` a v infikovaných boot sektorech byla nalezena zpráva `"Welcome to the Dungeon"`. Ačkoliv tento virus nevyužíval žádnou techniku k zašifrování nebo změně svého těla, odstartoval velkou éru polymorfních a metamorfních virů, které uměly měnit svůj kód a lépe se bránit proti detekci antivirovým programem. [4]

První malware, který šifroval své tělo byl virus Cascade z roku 1987. Infikoval soubory s příponou `.com` a způsoboval efekt padajících písmenek po obrazovce. K šifrování svého těla využíval symetrickou XOR šifru, kde klíčem byla velikost infikovaného souboru.

2.4 Současná situace

Časy, kdy byl malware vyvíjen převážně teenagery jako žertík, nebo pro ukázání svých dovedností, jsou dávno pryč[5]. Dnešní malware je vytvářen především za účelem krádeže informací nebo peněz. Jsou implementovány v různých programovacích jazycích s různými kompilátory, a jejich kód je maskován nejrůznějšími metodami ke stížení jejich detekce a zpětné analýzy.

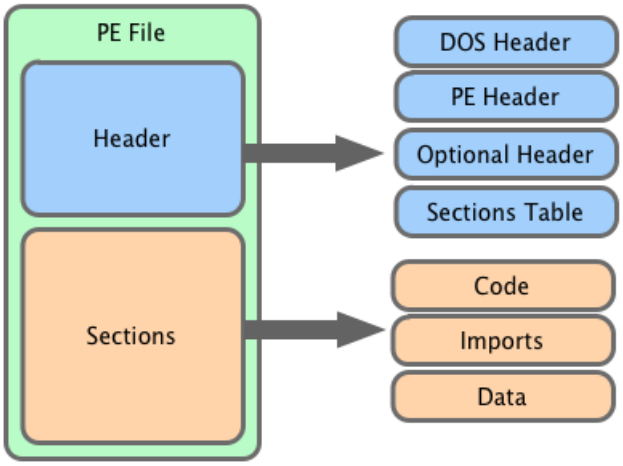
V současné době je malware maskován nejčastěji pomocí packerů, které nabízí aplikaci kompresních, šifrovacích a obfuskačních metod na spustitelné PE soubory.

Metody detekce porovnáváním kódu virů, které využívaly někdejší antivirové programy lze dnes snadno obejít obfuskací a kompresí. To, co však nelze změnit je chování viru. Společnost ESET[5] v současnosti využívá DNA detekce, které pracují s definicemi chování a charakteristickými vlastnostmi škodlivého kódu. Pomocí hloubkové analýzy kódu extrahují tzv. "geny", které představují určitou charakteristiku chování. Díky tomu lze snadněji odhalit škodlivý kód na disku nebo v paměti, a dokonce detekovat i jeho modifikované nebo doposud neznáme varianty, které prokazují známky stejného chování.

Společnost Avast[6] využívá podobnou technologii zvanou DeepScreen. Tato technologie využívá algoritmů strojového učení, díky kterým je schopna identifikovat podobnosti s již známým malwarem a následně jej deobfuskovat.

2.5 Portable Executable

Dříve, než budou v následujících kapitolách představeny jednotlivé kompresní, šifrovací a obfuskační techniky a jejich detekce, je nejprve potřeba porozumět konceptu formátu PE (angl. Portable Executable). Jedná se o nativní formát pro spustitelné soubory (.exe), dynamické knihovny (.dll) a objektové soubory používaný v 32bit a 64bit verzích operačního systému Windows. Je to datová struktura obsahující všechny informace potřebné pro zavaděč OS Windows, jak s daným kódem programu zacházet. Soubor PE formátu se skládá ze dvou hlavních částí - hlaviček a sekcí. V hlavičkách jsou obsaženy detailní metadata o samotném PE souboru a v sekcích je pak obsah daného programu. Na obrázku 1 lze vidět kompletní strukturu PE souboru.



Obrázek 1: Struktura souborového formátu PE (Převzato z [14])

2.5.1 Hlavička MS-DOS MZ

Začátek spustitelného souboru EXE začíná hlavičkou MS-DOS (MS-DOS stub). Jedná se o program spustitelný pod MS-DOS a pokud je spuštěn pod tímto operačním systémem, vypíše hlášku *"This program cannot be run in DOS mode"*. Důležitou částí této hlavičky jsou první dva bajty, které mají hodnotu "4D 5A"(hexadecimálně), což v ASCII představuje znaky "MZ", které představují iniciály tvůrce tohoto formátu, Marka Zbikowskiho.

2.5.2 Hlavička PE

Za hlavičkou MS-DOS MZ následuje hlavička PE na offsetu 0x3C. Jedná se o 4 bajtový podpis s hodnotou "PE\0\0"(znaky 'P', 'E' a dva nulové bajty), který nám říká, že daný soubor je PE formátu pro OS Windows.

2.5.3 Nepovinná hlavička (Optional header)

Tato hlavička je nepovinná pouze u objektových souborů. U spustitelných souborů a dynamických knihoven je však povinná. Existují dvě verze optional headeru - pro 32bit a 64bit architekturu. Obě však nesou důležité informace, jako např. velikost sekce s kódem, velikost zásobníku, haldy, atd. Velmi důležitou součástí této hlavičky je ukazatel na tzv. original entry point (OEP), což je adresa, na které se začíná program spouštět. Tato adresa bývá často zaměňována packery za účelem obfuskace a komprese, o nichž bude zmíněno v následujících kapitolách.

2.5.4 Tabulka sekcí

Tabulka sekcí se skládá z hlaviček sekcí *(section headers)*, což jsou reference na jednotlivé sekce obsažené v PE souboru. Jednotlivé hlavičky obsahují jméno sekce, její adresu na umístění v souboru, velikosti na disku a další charakteristiky.

2.5.5 Sekce PE formátu

Každá ze sekcí definovaných v tabulce sekcí obsahuje informace o funkcionalitě daného programu, které si operační systém následně načítá a zpracovává do strojového kódu. Mezi nejčastěji využívané sekce PE soboru patří[15]:

- **.text** - Obsahuje strojový kód programu, který procesor následně zpracovává. Jedná se o jedinou sekci, ze které je program spouštěn a obsahuje již dříve zmíněný original entry point (OEP), jako počáteční bod spouštěného kódu .
- **.rdata** - Obsahuje data, která slouží pouze pro čtení a jsou dostupná globálně a program k nim může přistupovat. Důležitou součástí této sekce bývají také informace o importovaných funkcích z jiných DLL knihoven, a exportovaných funkcích, které jsou zase importovány jinými PE soubory (obvykle jsou přítomny pouze u DLL knihoven, protože spustitelné

soubory EXE většinou neexportují žádné funkce). V některých případech nejsou informace o importech a exportech obsaženy v této sekci, nýbrž v dalších sekcích `.idata` (importy) a `.edata` (exporty).

- **.data** - Obsahuje globální data přístupné kdekoliv z aplikace
- **.rsrc** - Obsahuje všechny zdrojová data *(resources)* jako například obrázky, ikony, zvuky, nebo řetezce používané v programu.
- **.reloc** - Obsahuje informace o adresách k realokaci v případě, že operační systém nemůže program načíst na adresu specifikovanou v PE hlavičcce (jelikož jsou na tuto adresu již načteny jiná data).

2.6 Šifrování

Nejstarší a zároveň nejjednodušší metodou k zamaskování těl virů je šifrování. Takový šifrovaný virus se skládá ze dvou částí: decryptoru a zašifrovaného těla viru. Decryptor je ta část kódu, kterou kód viru obvykle začíná a provádí šifrování a dešifrování zbytku těla. Zašifrované tělo je tak pro reverzního inženýra nečitelné, dokud není dešifrované. Jakmile je program infikovaný šifrovaným virem spuštěn na hostitelském počítači, decryptor nejprve dešifruje zbytek těla viru do spustitelné podoby. Důležitým poznatkém je, že dešifrované tělo viru je čitelné pouze v operační paměti počítače. Pokud chce tento virus infikovat další program, nejprve znovu zašifruje své tělo a společně s decryptorem se připojí k hostitelskému programu. Většina šifrovaných virů používá při každém novém připojení na hostitelský program odlišný klíč. To znamená, že šifrované tělo viru je při každé další infekci odlišné a stejný zůstavá pouze decryptor. [3]

Šifrované viry přináší pro antivirové programy řadu komplikací. Jelikož je tělo viru při každé další infekci zašifrováno jiným klíčem, není tak snadné jej porovnat s již známými viry z virové databáze. Šancí na detekci takového viru je detekce podle jeho decryptoru, protože ten zašifrován nikdy není. Avšak i to není úplně spolehlivá metoda, protože stejný decryptor může používat spousta dalších virů, takže antivirus nemusí detekovat správnou variantu. Stejný decryptor můžou použít dokonce i nevirové programy, které se takto brání proti ladění.

2.6.1 XOR šifrování

Šifrování těla viru může být provedeno několika metodami. Nejčastější z nich je použití operace XOR s každým bajtem kódu viru. XOR je velmi rychlá a oboustranná šifra, a proto je mezi viry velmi oblíbená. Není tak potřeba implementovat dva rozdílné algoritmy pro šifrování a dešifrování. Prvním zašifrovaným virem šifrou XOR byl v historii právě již zmíněný virus Cascade z roku 1987 určený pro MS-DOS.

```
lea   si,Start    ; začátek zakódovaného těla (nastaveno virem)
mov   sp,0682h    ; délka zakódovaného těla (1666 bajtů)
Decrypt:
xor   [si],si     ; XOR šifrování
xor   [si],sp     ; XOR šifrování
inc   si          ; inkrementace ukazatele
dec   sp          ; dekrementace čítače bajtů
jnz   Decrypt     ; proveď pro další bajt (dokud nejsou dešifrovány všechny)
Start:            ; začátek virového kódu
```

Výpis 1: Ukázka decryptoru viru Cascade v assembleru (převzato z [1])

Na decryptoru viru Cascade lze vidět, že nemá adresu na začátek zašifrovaného těla viru určenou kompilátorem. Tato adresa je určena během infekce na základě pozice virového těla v infikovaném souboru. V případě Cascade se virové tělo připojuje na konec, a tak bude registr SI, který zároveň slouží jako klíč pro dešifrování, mít hodnotu odpovídající velikosti infikovaného souboru. Pokud tedy dva různé soubory mají stejnou velikost, klíč bude stejný. Registr SP (ukazatel na zásobník) je zde použit jako čítač počtu bajtu k dešifrování (virus zná velikost svého těla ještě předtím, než infikuje hostitelský soubor). Takové použití ukazatele na zásobník je zároveň obranou proti ladění, protože jakékoliv jiné použití zásobníku (například breakpointem v kódu decryptoru) by zničilo data v paměti, kde ukazatel na zásobník odkazuje (čili tělo viru, které se decryptor snaží dešifrovat). Dešifrování operací XOR je prováděno tak dlouho, dokud nejsou dešifrovány všechny bajty virového těla.

2.6.2 Base64 šifrování

Base64 je systém šifrování, který převádí binární data do textové podoby. Používá se k přenosu binárních dat v systémech, přes které můžeme přenášet pouze textová data. Ve standardu MIME se k šifrování využívá abecedu 64 znaků složená z malých a velkých písmen anglické abecedy, číslic, znaků "+"a "/". Zašifrovaná data bývají obvykle delší než původní. Rozeznat, že jsou nějaká data zašifrována pomocí Base64 je jednodušší než u jiných šifrovacích systémů, a to zejména díky jednomu nebo dvěma rovnítkům, které se doplňuji na konec zašifrovaného řetězce, pokud počet oktetů původních binárních dat není dělitelný třemi. Autoři malwaru však často ještě zamění abecedu MIME standardu za svou vlastní, čímž znesnadní práci reverzním inženýrům, a standardizované dekódéry, které jsou často součástí různých nástojů operačních systémů, nemůžou takový malware dešifrovat správně.[7][8]

V současné době šifrují útočníci pomocí Base64 nejčastěji Powershell scripty. Pomocí parametru `-EncodedCommand` tak můžeme v Powershellu spustit příkazy, které jsou zašifrovány pomocí Base64, aniž bychom je předtím viděli dešifrované. Pokud chceme daný řetězec dešifrovat, Powershell k tomu poskytuje funkci `FromBase64String`. Je možné, že dešifrovaný script bude obsahovat další Base64 retězce. Pak se jedná o tzv. vícenásobné šifrování.[9]

```
# Spusteni zasifrovaneho prikazu
$command = 'dir "c:\program files" '
$bytes = [System.Text.Encoding]::Unicode.GetBytes($command)
$encodedCommand = [Convert]::ToBase64String($bytes)
powershell.exe -encodedCommand $encodedCommand
```

Výpis 2: Ukázka spuštění zašifrovaného Powershell příkazu

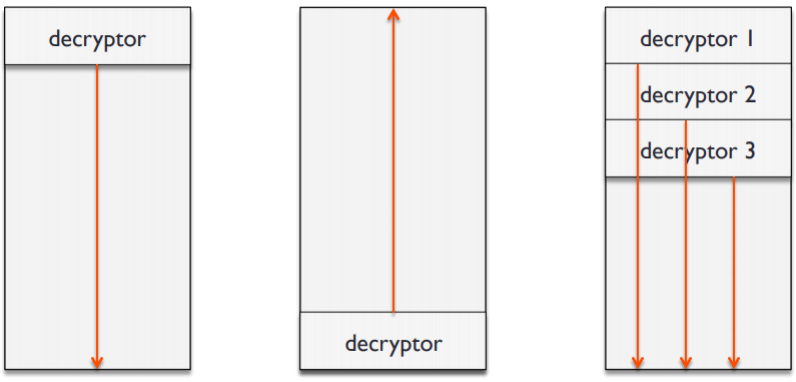
2.6.3 ROT13 šifrování

ROT13 je jedna z variant Caesarovy šifry, kdy je abeceda rozdělena do dvou řádků po 13 znacích a šifrování a dešifrování je prováděno záměnou písmen, které se nacházejí pod a nad sebou. Je velmi podobná šifře XOR, jelikož šifrování stejné hodnoty dvakrát pomocí ROT13 nám vrátí hodnotu původní. Používá se především k obfuskaci textových řetězců ve zdrojových kódech (např. URL). Útočník si stejně jako u Base64 může určit svoji vlastní abecedu, pomocí které bude šifrovat a dešifrovat, a tím tak ztížit případnou zpětnou analýzu.[7]

2.6.4 Pokročilejší způsoby šifrování

Někteří útočníci v dnešní době využívají spoustu sofistikovanějších metod k zašifrování těla svých škodlivých programů. Častým případem je použití vícenásobného šifrování, kdy virus obsahuje více než jeden decryptor. Příkladem je třeba aplikace dvou decryptorů, kdy jeden dešifruje virové tělo směrem dopředu pomocí jednoho algoritmu, a následně druhý decryptor jej dešifruje směrem zpátky úplně jiným algoritmem. V jiném případě může první decryptor dešifrovat druhý nebo úplně jiný decryptor, který pak nakonec dešifruje virové tělo.[1] [11]

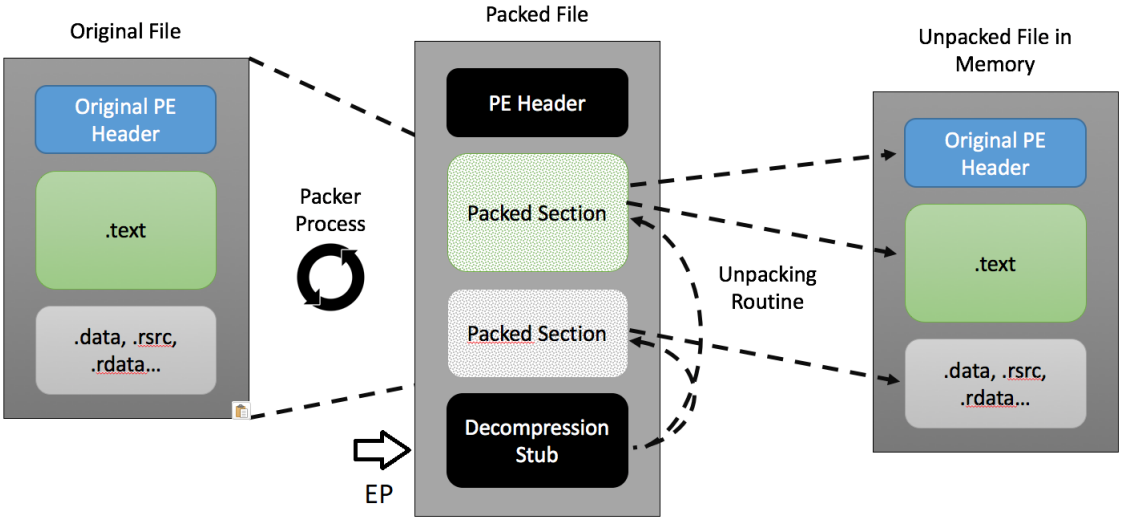
Útočníci také často využívají více než jeden klíč k dešifrování všech částí viru (i v případě použití pouze jednoho decryptoru). Obecně platí, že čím delší klíč je použit, tím déle bude trvat dešifrování hrubou silou, a to i v případě použití více klíču může zpětnou analýzu ještě více znesnadnit.



Obrázek 2: Ukázka různých způsobů implementace decryptoru (převzato z [10])

2.7 Komprese

Komprese je způsob transformace spustitelného souboru, při které dochází k jeho zmenšení a zároveň znečitelnění původního kódu. Dochází k vytvoření komprimovaného souboru, který ukládá původní soubor jako data (často v přídavných sekcích). Zároveň je k souboru připojen tzv. dekompresní algoritmus (*unpacking/decompression stub*), na který je nastaven EP. Komprimovaný kód nelze přímo spustit operačným systémem, dokud není provedena jeho dekomprese pomocí dekompresního algoritmu. Během spůštění komprimovaného programu se nejprve spustí dekompresní algoritmus, který dekomprimuje původní kód programu. Po dekompresi je vykonávání kódu nastaveno již na začátek původního kódu (OEP). Původní kód programu je tedy viditelný až v operační paměti (pokud tedy soubor nedekomprimujeme potřebným algoritmem ještě dříve, než je spuštěn). Komprese je také nazývana jako tzv. *packing* nebo *executable compression*, a je nejčastěji prováděna automaticky pomocí programů nazývané jako packery. Komprese je v dnešní době společně s šifrováním a obfuskací velmi oblíbenou metodou maskování těl malwaru. Statisticky je dokázáno, že 80-90% malwaru je komprimovaných. Na obrázku 3 můžeme vidět, jak komprese a následná dekomprese PE souboru probíhá.

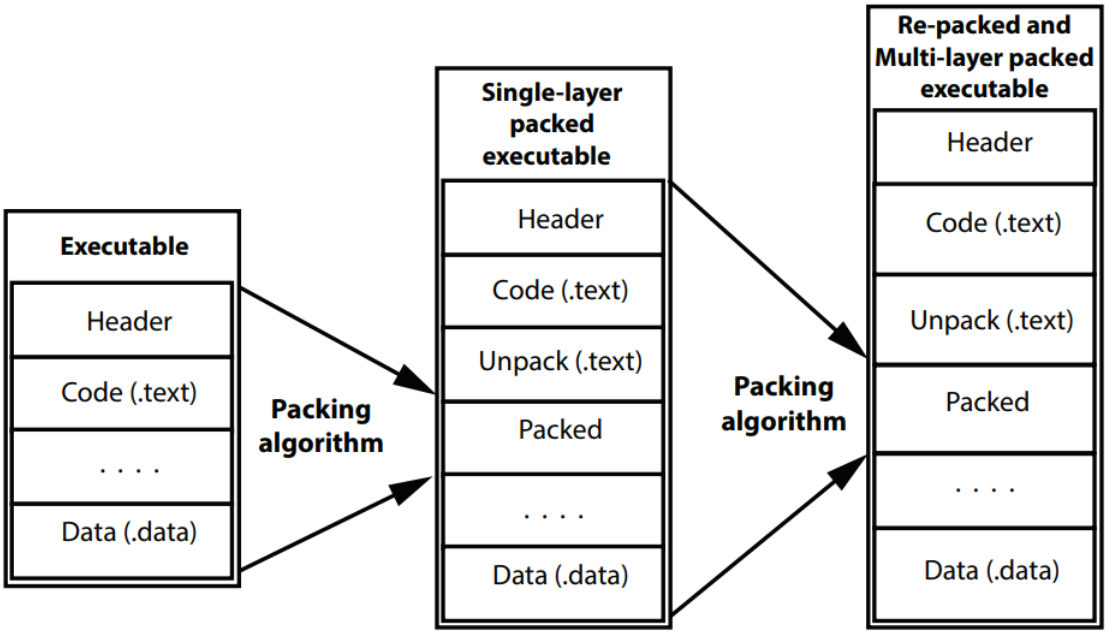


Obrázek 3: Ukázka průběhu komprese a dekomprese PE souboru (převzato z [12])

2.7.1 Vícenásobná komprese

PE soubor lze komprimovat i více než jedním packerem. Při použití dalšího packeru je vždy původní komprimovaný soubor uložen opět jako data v nově komprimovaném souboru. Při spuštění je soubor dekomprimován postupně od posledně použitého packeru až po první. V současnosti je vícenásobná komprese často používána, avšak ne mezi běžně dostupnými packery, jelikož nejsou mezi sebou tolik kompatibilní, nebo si chrání soubor před použitím dalšího packeru. Autoři ma-

lwaru si k vícenásobné kompresi nejčastěji vytvářejí vlastní algoritmy a packery. Na obrázku 4 můžeme vidět průběh vícenásobné komprese PE souboru.



Obrázek 4: Ukázka průběhu vícenásobné komprese PE souboru (převzato z [13])

2.8 Packery

Za packer se dá považovat software, jehož vstupem je soubor PE formátu a výstupem nový soubor PE formátu, který je komprimovanou, šifrovanou nebo obfuskovanou verzí souboru vstupního. Ke komprimovanému souboru je zároveň přidán dekompresní algoritmus, který jej dekomprimuje při jeho spuštění. Spousta packerů ještě navíc upravuje vstupní soubor tak, aby jej nešlo disassemblovat, debugovat, nebo spouštět ve virtuálních počítačích.

2.8.1 Klasifikace packerů

Většina packerů pro PE formát pracuje jen s EXE spustitelnými soubory a DLL dynamickými knihovnami. Podle druhu funkcionalit lze packery rozdělit do čtyř hlavních kategorií:

- **Kompresory** *(Compressors)* - Kompresory zmenšují velikost PE souborů pomocí kompresního algoritmu. Jakmile je komprimovaný soubor spuštěn, je dekomprimován přímo do paměti. Někdy se tato technika nazývá také jako "spustitelná komprese"nebo "samorozbalovací archívy". Původně byly kompresory vytvořeny pouze za tímto účelem, avšak se stále narůstajícími velikostmi paměťových médií se v dnešní době využívá spíše jen u malwaru, protože výstupní soubor je mnohen složitější analyzovat, dokud není spuštěn.

- **Kódery** *(Crypters)* - Kódery mají za úkol daný soubor zašifrovat pomocí různých šifrovacích algoritmů jako XOR, Base64 apod.
- **Protektory** *(Protectors)* - Protektory nejčastěji představují kombinaci kompresorů a kóderů. Nabízí však i další ochrany softwaru jako např. licencování, ochrana proti debugování, apod. Často také do výstupního souboru přidávají různá metadata nebo ikony, aby se program tvářil jako důvěryhodný.
- **Bundlery** *(Bundlers)* - Bundlery zabalují více spustitelných souborů a další datové soubory do jednoho spustitelného souboru, které není potřeba nijak extrahovat na disk, jakmile je takový soubor spušten.

2.8.2 UPX packer

UPX[25] packer se řadí mezi kompresory, je vyvíjen již od roku 1996 a dá se považovat za nejpopulárnější mezi všemi packery. Je zdarma, open-source pod licencí GNU v jazyce C++, takže je snadno přenositelný mezi řadou platforem. Mezi jeho hlavní přednosti patří především kvalitní kompresní poměr, který je typicky lepší než u kompresorů jako WinZip nebo GZip, vysoká rychlost dekomprese a nízká paměťová náročnost na dekompresi.

Jelikož je UPX packer velmi známý, není v dnešní době vůbec obtížné detekovat, že je pomocí něj daný soubor komprimován. Samotný packer zároveň umožňuje i dekompresi souboru do původní podoby, takže nám stačí pouze detekovat, že je daný soubor tímto packerem komprimován. UPX packer je velmi dobrým nástrojem pro naučení se, jak jsou soubory komprimovány a jak je lze manuálně detekovat a dekomprimovat. Spousta autorů malwaru však používá modifikované verze UPX packeru, takže dekomprese není pomocí původního originálního UPX možná. To často zkomplikuje práci reverzním inženýrům, jelikož komprimovaný soubor může při spuštění vykazovat jiné známky chování, než kdyby byl komprimovaný originálním UPX.

UPX packer komprimuje originální soubor spojením všech sekcí PE souboru do jedné s názvem `UPX1` Jedinou vyjímkou je sekce `.rsrc` obsahující zdrojová data *(resources)*. Kromě této sekce je v komprimovaném souboru obsažena také sekce `UPX0`, která je prádzná a rezervuje místo potřebné pro komprimovaná data, jakmile dojde k dekompresi při spuštění programu.

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers
000001C8	000001D0	000001D4	000001D8	000001DC	000001E0	000001E4
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword
UPX0	00008000	00001000	00000000	00000400	00000000	00000000
UPX1	00004000	00009000	00003600	00000400	00000000	00000000
.rsrc	00001000	0000D000	00000A00	00003A00	00000000	00000000

Obrázek 5: Seznam sekcí PE souboru komprimovaného pomocí UPX packeru

2.8.3 ASPack

ASpack[26] je komerční packer vyvíjený společností StarForce Technologies určený pro 32bit Windows PE soubory (EXE, DLL). Jeho kompresní poměr je až 70%. Metody na dekomprimaci souborů z výstupu tohoto packeru jsou však na internetu snadno dohledatelné a zároveň existuje spousta nástrojů, které dekompresi provedou automaticky. Vedle klasického ASpack packeru společnost StarForce Technologies nabízí i protektory ASProtect32 (pro 32bit platfotmy) a ASprotect64 (pro 64bit platformy). Tyto protektory umí spustitelný soubor zároveň šifrovat, chránit jej proti disassemblování a debugování.

2.8.4 PECompact

PECompact[27] je freeware packer společnosti Bitsum Technology. Dlouhodobým uživatelem PECompact packeru byla v minulosti společnost Google, která jej využívala ke kompresi svých desktopových aplikaci (např. instalační soubor Google Chrome). Používaný kompresní algoritmus je optimalizovaný právě pro PE soubory. Hlavní myšlenkou PECompactu je omezit jeho použití hackery a přinášet benefity spíše pro běžné uživatele a firmy. Tato myšlenka je realizována vysokou spoluprací s antivirovými společnostmi, kterými jsou předkládány informace o technikách používaných ke kompresi PE souborů. Antivirové programy tak posléze snadno detekují soubor komprimovaných PECompactem a nedochází k tak častým FP *(false-positive)* detekcím. Narozdíl od jiných packerů, PECompact při kompresi nevytváří žádné další přídavné sekce s nestandartními názvy. Všechny sekce jsou ponechány a komprese kódu je provedena přímo v sekci `.text`. PECompact nabízí robustním plug-in SDK pro vytváření zásuvných modulů. V kompresním poměru je považován za kvalitnější než ASPack a stejně kvalitni jako UPX. Nevýhodou tohoto packeru je, že pracuje pouze s 32bit PE soubory (EXE a DLL). PE+ (64bit PE) nebo .NET spustitelné soubory a knihovny nejsou podporovány. Je možné jej spouštět jak pomocí grafického rozhrání, tak přes konzoli.

2.8.5 PELock

PELock Software Protection [28] je protektor pro 32bit PE soubory. Kromě komprese nabízí velkou řadu dalších funkcionalit k zabezpečení softwaru. Jednou z nich je zabudovaný licenční systém, díky kterému můžeme aplikaci zabezpečit pomocí licenčního klíče, nebo např. 30 denní trial lhůtou. PELock však hlavně nabízí vysoce kvalitní ochranu před zpětnou analýzou kódu, a to zejména díky použití složitých šifrovacích technik. Využívá technologii zvanou *"Polymorphic Engine"*, která při každém šifrování využívá jiný algoritmus sestavených z náhodně vybraných operací pro šifrování. Decryptor přidaný k aplikaci pro dešifrování je vysoce propojen s kódem aplikace, takže pokud se útočník staží odstranit nebo jakkoliv modifikovat decryptor, kód aplikace se může snadno poškodit a stát nespustitelným. PELock mimo jiné také nabízí pokročilé ochrany jako např. emulaci funkcí Windows API namísto jejich přímého volání, ochranu proti debugování nebo spouštění decryptoru ve více vláknech. Stejně jako u PECompactu je PE-

Lock kompatibilní s většinou známých antivirových programů, čímž se snižuje šance detekování chráněného softwaru jako FP *(false-positive)*.

2.8.6 Obsidium

Obsidium[29] je protektor pro 32bit i 64bit PE a .NET spustitelné soubory. Mezi jeho největší funkcionality patří především virtualizace kódu, kde je nativní kód aplikace převeden do bajt-kódu pro virtuální CPU, který je interpretován za běhu aplikace. Další velkou funkcionalitou je dešifrování konkrétních částí kódu programu až v paměti počítače, a to při jejich vykonání. Obsidium, stejně jako PELock, nabízí systém licencování, ochranu softwaru heslem, apod.

2.8.7 Srovnání packerů podle výskytu funkcionalit

Na základě výše popsaných packerů vznikla tabulka srovnávající jednotlivé packery dle výskytu vybraných funkcionalit. Jak lze v tabulce 1 vidět, protektory (PELock, Obsidium) často nabízí více možností k ochraně softwaru, které si může uživatel navolit, než packery UPX, ASPack a PECompact, které se soustředí více na kvalitu komprese. Nejmenší kompatibilita s antivirovými programy je u packeru Obsidium, jelikož není tolik známý a tak hojně používaný.

Tabulka 1: Srovnání packerů podle výskytu funkcionalit

	UPX	ASPack	PECompact	PELock	Obsidium
Komprese	✓	✓	✓	✓	✓
Šifrování	✗	✗	✗	✓	✓
Obfuskace	✗	✗	✗	✓	✓
Antidebugging	✗	✓	✗	✓	✓
Plu-gin SDK	✗	✗	✓	✓	✓
Podpora PE+ (PE 64bit)	✗	✗	✗	✗	✓
Podpora .NET assembly	✗	✗	✗	✗	✓
Kompatibilita s AV programy	✓	✓	✓	✓	✗

2.9 Detekce komprimovaných souborů

2.9.1 Obecné známky komprimovaných souborů

Každý packer komprimuje soubory jinými algoritmy. Některé kompresní algoritmy jsou známy (např. UPX packer), avšak útočníci si často vytváření své vlastní algoritmy, nebo modifikují již existující. Většina komprimovaných souborů má však společné znaky. Zde je seznam známých heuristických metod k nalezení komprimovaného souboru[2]:

- Nestandartní názvy sekcí - Komprimované soubory často obsahují sekce s jinými názvy (např. u UPX packeru sekce `UPX0` a `UPX1`) a původní sekce (`.text`, `.rdata`, atd.) nemusí být vůbec obsaženy.
- Počet sekcí s povoleným zápisem - Komprimované soubory musí mít alespoň jednu sekci s povoleným zápisem (u UPX např. UPX0, do které se vkládají data při dekompresi). Nekomprimované soubory nemusí mít ani jednu sekci s povoleným zápisem.
- Počet importovaných funkcí Windows API - Komprimované soubory obsahují méně importovaných funkcí než nekomprimované. Často to bývají pouze funkce `LoadLibrary` a `GetProcAddress`.
- Vysoká entropie - Entropie komprimovaného souboru bývá vysoká (v průměru 6,8 a více), jelikož data obsažená v souboru vypadají jako náhodná.
- Velikost sekcí - V komprimovaných souborech můžou být obsaženy sekce s nulovou velikostí na disku *(raw size)* a nenulovou virtuální velikostí (velikost sekce při nahrání do paměti)
- Při otevření souboru v OllyDbg vyskočí hláška, že soubor může být komprimovaný.

2.9.2 Detekce pomocí entropie

Bajtová entropie je v informatice míra neuspořádanosti digitálních dat v nějakém systému nebo souboru. Je to hodnota od 0 do 8, kde vyšší číslo znamená větší neuspořádanost. Kompresní a šifrovací metody transformují originální bajty souboru do série náhodně uspořádaných bajtů, a proto komprimované, šifrované nebo obfuskované soubory mají vyšší entropii, než soubory originální. Entropii dat lze zpočítat podle následujícího Shanonova vzorce [24]:

$$H(x) = -\sum_{i=1}^{n} p(i) \log_2 p(i),$$

kde $H(x)$ je hodnota měřené entropie, $P(i)$ je pravděpodobnost výskytu i-té jednotky informace (v našem případě bajtu) v sérii n jednotek náhodného jevu x.

Pomocí entropie tak můžeme snadno zjistit, zda je daný soubor komprimovaný, šifrovaný nebo obfuskovaný. Měření lze provádět pro celý soubor, ale i pro každou sekci PE souboru zvlášť. Například u UPX packeru lze zaznamenat vysokou entropii v sekci `UPX1`, která obsahuje komprimovaný kód původního programu (původně sekce `.text`, `.data`, atd.). Hodnota entropie pro komprimované, šifrované nebo obfuskované soubory se pohybuje okolo hodnot 6,8 - 8. [24]

2.9.3 Detekce pomocí signatur

Detekce pomocí signatur je známou metodou, kterou antivirové programy využívají k detekci virových souborů. Pro packery lze tuto metodu použít také, kde každá signatura představuje

určitý packer konkrétní verze. Kód zkoumaného programu je analyzován a pokud je v něm nalezena nějaká signatura z databáze, soubor je označen jako komprimovaný packerem, ke kterému signatura náleží. Známé detektory pracující na tomto principu jsou PEiD a ExeInfo PE, které jsou podrobněji popsány v kapitole 2.10. Na obrázku 6 můžeme vidět signaturu pro packer UPX z databáze pro detektor PEiD. za otázníky lze dosadit libovolný bajt.

```
8759 [Simple UPX Cryptor V30.4.2005 -> MANtiCORE]
8760 signature = 60 B8 ?? ?? ?? ?? B9 ?? ?? ?? ?? ?? ?? ?? ?? E2 FA 61 68 ?? ??
     ?? ?? C3
```

Obrázek 6: Signatura packeru UPX

2.9.4 Detekce pomocí strojového učení

Algoritmy umělé inteligence a strojového učení jsou v poslední době čím dál více využívány v oblasti počítačové bezpečnosti. Pokud jsme schopni nasbírat větší množství dat o PE souborech, můžeme jimi natrénovat klasifikátor. Pomocí binární klasifikace jsme schopni rozeznávat, zda je soubor komprimovaný či nikoliv. Vícetřídní klasifikace nám může dokonce pomoci rozeznat, který packer byl na zkoumaný soubor použit (ovšem jsme omezeni pouze na packery, které jsou obsaženy v datovém setu). Praktické provedení detekce pomocí strojového učení je popsáno v kapitolách 3.4.2 a 3.5

2.10 Nástroje pro detekci packeru

Při analýze malwaru chceme nejprve zjistit zda je daný program komprimovaný nebo jinak chránený, a ideálně se rovnou dozvědět, který packer byl k tomu použit. Existuji řada nástrojů, které dokážou packer detekovat. Tyto nástroje pracují na metodě vyhledávání již známých signatur kódu v těle analyzovaného programu, které jsou pro každý packer nějak specifické. Některé z nich i vypočtou entropii komprimovaného souboru nebo dokonce provedou dekomprimaci. Mezi nejznámější nástroje pro detekci packeru patří PEiD, EXEinfo PE a DiE.

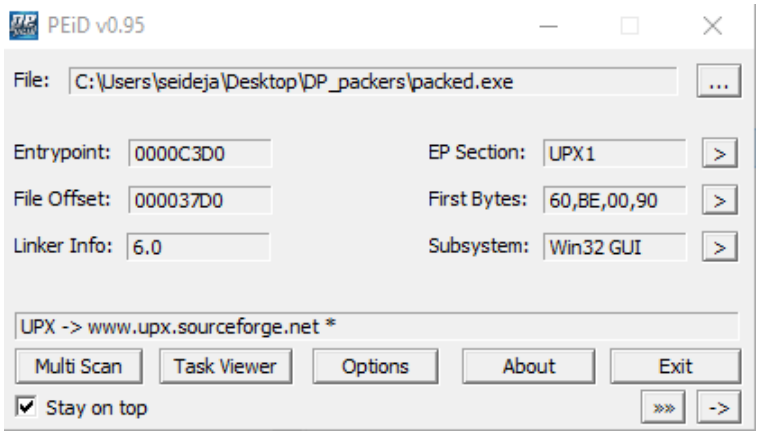
2.10.1 PEiD

PEiD[30] je nejznámější nástroj pro detekci packeru, kóderu a kompilátoru pro PE soubory. Ačkoliv je jeho vývoj již ukončen a nejvyšší verze je 0.95, obsahuje databázi více než 600 signatur různých packerů, které jsou následně v analyzovaných PE souborech vyhledávány.

PEiD nabízí tři módy skenování PE souborů. Normal mód vyhledává známé signatury okolo entry point (EP). Deep mód skenuje navíc celou sekci s EP obsahující komprimovaný kód původního souboru, což zajistí detekci až 80% souborů, které jsou komprimované, či jinak chráněné. Hardcore mód skenuje kompletně celý soubor, což může trvat výrazně delší čas, než u dvou předchozích. Všechny skenovací techniky jsou ošetřeny metodou kontroly chybových výstupů

(chybná detekce packeru), které můžou nastat u menších signatur, jelikož se často vyskytují i ve spoustě jiných souborů, které nejsou nijak komprimované.[31]

Mezi hlavní funkcionality PEiD kromě detekce packeru patří zobrazení seznamu jednotlivých sekcí, importů a exportů, jednoduchý disassembler, výpočet entropie, skenování aktuálně souštěných procesů nebo celé složky. V neposlední řadě obsahuje také rozhraní pro pluginy, kterých zdarma ke stažení existuje na internetu celá řada. Jedná se hlavně o pluginy pro dekompresi. Samotná výchozí verze PEiD má již v sobě obsažené pluginy jako "Generic OEP Finder"(generický vyhledávač OEP), "Krypto Analyzer"(vyhledávání zašifrovaných řetězců) a "Generic unpacker"(generický unpacker).[32] Na obrázku 7 můžeme vidět GUI PEiD.

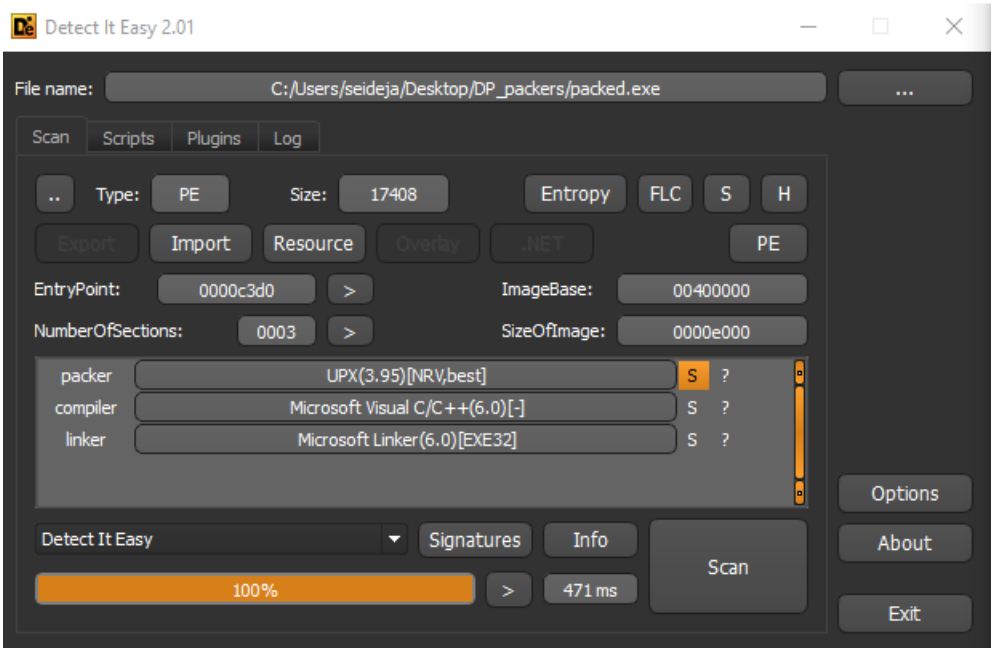


Obrázek 7: Detekování UPX packeru pomocí PEiD

2.10.2 DiE (Detect it Easy)

Detect it Easy[33] je ruský multiplatformní nástroj pro analýzu souborů. Kromě souborů PE pro Windows umí analyzovat i spustitelné soubory ELF (Linux) a MACH-O (Mac OS). Narozdíl od PEiD, který obsahuje databázi signature ve formě masek bajtů, DiE obsahuje databázi algoritmů pro detekci jednotlivých packerů, kóderů, kompilátorů a linkerů ve formě scriptů. Uživatel si tak může napsat nový algoritmus pomocí scriptovacího jazyku velmi podobnému JavaScriptu. Nabízí se mu tak širší možnosti definování dalších parametrů k detekci, než jen masky bajtu, jak je tomu u PEiD.[34]

DiE přichází také s mnohem větší škálou funkcionalit a rozšířenějším uživatelským rozhraním než u PEiD. Mezi jeho přednosti oproti PEiD patří zejména kvalitnější znázornění výpočtu entropie ve formě grafů, uživatelsky přívětivější disassembler nebo seznam textových řetězců a URL v dekomprimovaném souboru. Mezi hlavní nevýhodu DiE patří malá škála pluginů a nekompatibilita s pluginy PEiD. Na obrázku 8 můžeme vidět GUI DiE.



Obrázek 8: Detekování UPX packeru pomocí DiE

2.10.3 Exeinfo PE

Exeinfo PE je velkou obdobou PEiD, avšak s mnohem více funkcionalitami. Je stále ve vývoji a je zpětně kompatibilní s pluginy pro PEiD. V současné době se jedná o jedničku mezi analyzátory PE souboru a detektory packerů. Obsahuje databázi více než 1000 signatur 32bit souborů a přes 70 signatur 64bit souborů. Nevýhodou je, že nedetekuje starší packery. K tomuto účelu je však doporučeno použít PEiD.

2.11 Dekomprese

Dekomprese komprimovaného souboru je důležitou součástí pro statickou i dynamickou analýzu malwaru při hledání potencionálního payloadu. Jedná se v podstatě o odstranění vrstvy (nebo více vrstev v případě vícenásobné komprese) komprimovaného kódu programu a zpětné získání jeho původní podoby[16]. Obecně lze techniky dekomprese rozdělit do 2 kategorií:

- **Manuální** - Je prováděna malwarovými analytiky manuálně pomocí debuggerů jako OllyDbg nebo IDA Pro, ve kterých aplikaci debugují a snaží se analyzovat její chování, zejména chování dekompresního algoritmu. Tento proces je poměrně časově náročný a vyžaduje vysokou znalost assembleru. V případě nasazení zkušených analytiků se však může jednat o nejlepší a nejspolehlivější metodu, zejména pokud je zkoumaný virus komprimovaný sofistikovanějšími způsoby, na které už automatické nástroje nestačí.
- **Automatická** - Jedná se o automatizování procesu dekomprese programem zvaným jako unpacker. Automatická dekomprese se ještě dále rozděluje jako:

Statická - Unpacker je určen k dekompresi pouze specifického packeru v jeho konkrétních verzích bez toho, aniž by komprimovaná aplikace musela být spuštěna nebo emulována. Je předem známo a prozkoumáno, jak se packer chová, a na základě toho je vyvinut algoritmus pro dekompresi.

Generická - Jelikož hackeři často využívají vlastních packerů nebo modifikovaných verzí běžně dostupných, vznikají univerzální řešení, která se snaží dekomprimovat většinu z nich. Takové unpackery se nazývají jako generické. V tomto případě je však potřeba komprimovanou aplikaci spustit nebo emulovat, dokud není plně dekomprimována v paměti počítače, ve které je následně vyhledáván originální kód. Po nalezení originálního kódu v paměti se unpacker pokusí o rekonstrukci původního programu. V současné době antivirové programy generickou dekompreci příliš nevyužívají, protože se ne vždy jedná o spolehlivou metodu. Největší problém generické dekomprese je ten, že nikdy přesně nevíme, zda je soubor plně dekomprimován, či nikoliv. K analýze sofistikovanějších virů se složitější a vícenásobnou kompresí je lepší provést manuální dekompresi.

Během dekomprese (ať už automatické či manuální) se nám zpravidla nemusí podařit získat originální soubor v naprosto stejné podobě, jako byl předtím (vyjímkou jsou některé statické unpackery, které soubor dekomprimují do přesně původního stavu). Může se i stát, že dekomprimovaný soubor nebude možné spustit. V případě, že nepotřebujeme program spouštět, ale pouze jej staticky analyzovat, nám to však tolik nevadí. Stačí pouze získat původní kód programu. Obecně je pro dekompresi potřeba provést následující:

1. **Získání původního OEP** - Během komprese je EP nastaven na dekompresní algoritmus a my tak nevíme, kde začíná původní kód programu. Při spuštění programu nejprve dekompresní algoritmus dekomprimuje původní kód a po jeho vykonání je provedena instrukce `JMP` na adresu dekomprimovaného kódu. Právě tam, kde začíná dekomprimovaný kód, se nachází OEP.

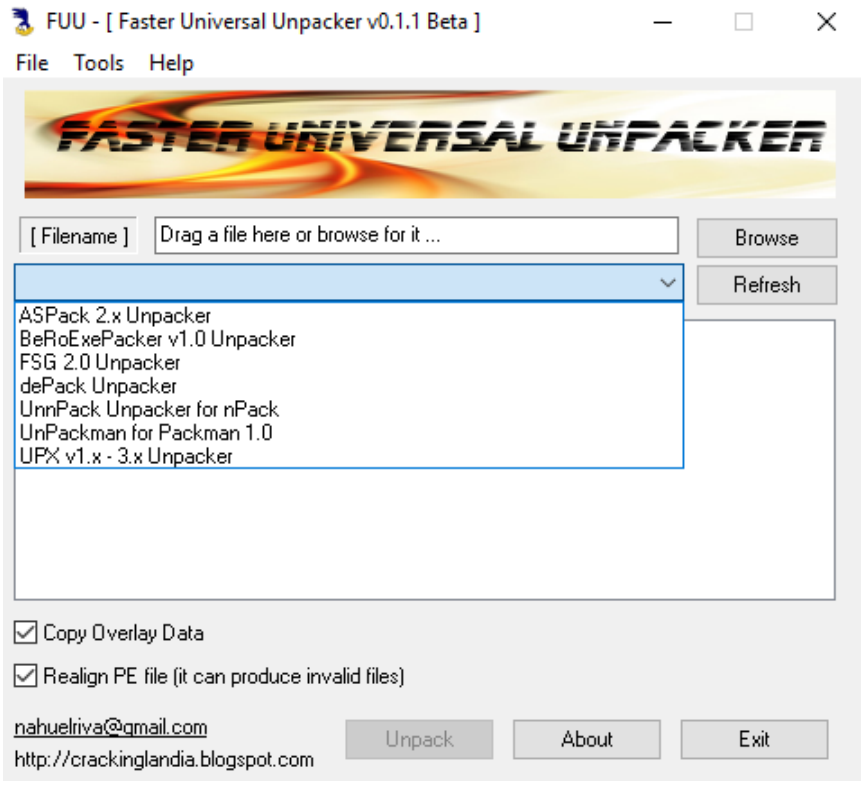
2. **Rekonstrukce tabulky importů** - Pokud chceme získaný soubor spouštět, budeme muset rekonstruovat i tabulku importovaných funkcí Windows API. Bez správných referencí na importy bude sice soubor možné staticky analyzovat, ale nepůjde spustit.

2.12 Nástroje pro automatickou dekompresi

Nástrojů pro automatickou dekompresi (unpackerů) existuje několik desítek až stovek. Ovšem sehnat unpacker funkční a kompatibilní s aktuálními verzemi nejznámějších packerů není tak jednoduché. V následujících dvou kapitolách budou představena současně nejznámější a nejdostupnější řešení pro dekompresi.

Pro statickou dekompresi:

- **Faster Universal Unpacker (FUU)** - FUU[19] je open-source unpacker s GUI (obrázek 9) podporující více packerů formou pluginů, které jsou naprogramovány v čistém assembleru. Využívá TitanEngine SDK (knihovna pro disassemblování, debugování a dump paměti analyzovaného PE souboru), díky kterého si lze naprogramovat vlastní plugin. V základní instalaci je obsaženo 7 pluginů pro dekompresi, včetně UPX a AsPack. FUU kromě dekomprese nabízí také detekci packeru, která funguje stejně jako v PEiD nebo ExeInfo a využívá jejich databází signatur. Nevýhodou tohoto unpackeru je, že od už roku 2013 není v aktivním vývoji, nicméně to nijak nenarušuje možnost tvorby nových pluginů.



Obrázek 9: GUI unpackeru FUU s rozbalenou nabídkou pluginů pro dekompresi

- **AspackDie** - Jde o jeden z nejznámějších unpackerů pro packer AsPack, který je kompatibilní i s jeho novějšími verzemi. Nevýhodou tohoto unpackeru je, že nedokáže obnovit tabulku importů, a tak je potřeba ji obnovit manuálně nebo pomocí jiných nástrojů.

- **UPX Unpacker** - Packer UPX[25] je známý tím že je open-source, a tak je jeho kompresní algoritmus známý. Vývojáři tohoto packeru s ním dodávají i funkcionalitu pro dekompresi, kterou lze v konzoli spustit příkazem `upx.exe -d [file]`. Soubor je následně dekomprimován do originálního stavu v podobě 1:1.

Pro generickou dekompresi:

- **Generic OEP Finder** - Jedná se o plugin pro aplikaci PEiD, který se snaží nalézt OEP. Tento plugin je kompatibilní i s aplikací ExeInfo PE. Analyzovaný soubor je během procesu vyhledávání spuštěn, a proto je při práci se skutečným malwarem doporučeno pracovat v odděleném prostředí.

- **PEiD Generic Unpacker** - Jde opět o plugin pro aplikaci PEiD, který se v první řadě snaží nalézt OEP a následně soubor dekomprimuje. Dekomprimovaný kód je uložen do nového souboru. Tento plugin je kompatibilní i s aplikací ExeInfo PE. Analyzovaný soubor je během procesu vyhledávání spuštěn, a proto je při práci se skutečným malwarem doporučeno pracovat v odděleném prostředí.

Následující řešení vycházejí z nalezených vědeckých článků zabývající se generickou dekompresí. Jedná se o pokusy vyvinout algoritmus, který by provedl co nejkvalitnější způsob dekomprese neznámého packeru, nikoliv o konkrétní software, který si lze stáhnout na internetu.

- **PolyUnpack** - PolyUnpack[17] je řešení založené na kombinaci statické a dynamické analýzy spustitelného souboru, které se snaží detekovat jeho dekomprimované části. Nejprve je vygenerován assemblerovský kód aplikace pomocí statické analýzy a následně je soubor spuštěň ve virtuálním prostředí, kde probíhá dynamická analýza. Během dynamické analýzy je spuštěný kód porovnáván s kódem z analýzy statické a jakmile se narazí na sekvence instrukcí, které ve statickém modelu chybí, je tento kód identifikován jako dekomprimovaný.

- **OmniUnpack** - Při spuštění komprimované aplikace dekompresní algoritmus nejprve dekomprimuje zbytek programu. Dekomprimovaný kód je dočasně uložen v paměti a následně spuštěn. OmniUnpack[18] využívá mechanismu pro sledování virtuální paměti a snaží se v ní detekovat takový kód.

2.13 Obfuskace

Obfuskace je taková transformace zdrojového kódu, která znesnadňuje jeho čitelnost a pochopení, aniž by byla narušena jeho původní funkčnost. Obfuskace může být aplikována po zkompilování daného programu packerem, který spustitelný soubor zároveň komprimuje případně šifruje, nebo ještě před kompilací na zdrojovém kódu. V následujících podkapitolách budou představeny některé obfuskační metody, které lze aplikovat na zdrojový kód. Ukázky obfuskovaného kódu jsou v jazyce C, případně v assembleru.

Podle teorie z roku 1997 vytvořenou americkým vědcem Christianem Collbergem [35] lze obfuskaci zdrojových kódů rozdělit do tří základních kategorií:

- Obfuskace dat
- Obfuskace běhu programu
- Obfuskace struktury programu

2.13.1 Hodnocení kvality obfuskace

Jelikož je obfuskace takový druh transformace softwaru, který nám jej má chránit proti reverzní analýze, ať už jsme autoři malwaru či jiného softwaru, bude nás jistě zajímat její kvalita. Kvalitu obfuskace určuje zejména její odolnost vůči automatickým deobfuskátorům a reverzním inženýrům, jak moc maskuje kód a jak moc zvyšuje výpočetní náročnost daného programu. Dříve než zde budou uvedeny konkrétní metody obfuskace softwaru, měli bychom si definovat několik základních parametrů, podle kterých můžeme následující metody a algoritmy ohodnocovat.

Kvalita konkrétní metody obfuskace je dána následujícími čtyřmi parametry[38]:

- **Účinnost** *(potency)* - Určuje, jak moc je obfuskovaný kód nerozeznatelný od originálního. Z pohledu složitosti kódu to může znamenat například počet použitých proměnných, hloubka dědičnosti nebo použití vnořených tříd a metod. Úkolem obfuskace je v tomto ohledu právě co nejvíce zvýšit složitost čtení kódu a tím pádem celkové architektury programu.
- **Odolnost** *(resilience)* - Určuje, jak moc je obfuskovaný kód schopen odolávat útokům automatických deobfuskátorů a reverzních inženýrů. V praxi můžeme odolnost chápat jako kombinaci času programátora potřebného k vytvoření funkčního automatického obfuskátoru, který prolomí danou obfuskaci, a dobu běhu samotného obfuskátoru. Zde je však nutné zmínit rozdíl mezi účinností a odolností, protože kód s vysoce účinnou obfuskací ještě nemusí být nutně odolný. Mějmě následující kusy originálního a transformovaného kódu:

```
// Originální kód
void main() {
   int n1 = 0;
   int n2 = 1;
}
```

```
// Transformovaný kód
void main() {
   int n1 = 0;
   if(1 > 2) n1 = 2;
   int n2 = 1;
   if(4 == 5) n2 = 3;
}
```

Výpis 3: Ukázka účinnosti obfuskovaného kódu

Na výpisu kódu č. 3 můžeme vidět, že kód je transformovaný přidáním podmínek, které nikdy nebudou pravdivé. Taková transformace tak zvyšuje kódu jeho účinnost, ale odolnost nikoliv, jelikož nedosažitelné podmínky můžou být snadno odstraněny automatickými technikami deobfuskátorů. Obfuskovaný kód je tedy účinný tehdy, když zmate člověka, a odolný tehdy, když deobfuskátor jej není možný vrátit do původního stavu. [38]

- **Utajení** *(stealth)* - Určuje, jak moc je obfuskace kódu spojena se zbytkem kódu programu. Pokud máme v kódu například metody, které provádějí obfuskaci ve speciálních třídách či dokonce knihovnách, může být pro deobfuskátor náročné je detekovat, avšak velmi snadné pro člověka (reverzního inženýra).
- **Náklady** *(transformation cost)* - Určuje navýšení výpočetního času a paměťové náročnosti programu s obfuskovaným kódem v porovnání s programem bez obfuskovaného kódu. Zde nám náklady na transformaci určuje asymptotická složitost obfuskovaného algoritmu v aplikaci. Příkladem může být přiřazení hodnoty do proměnné v programu. Pokud ji přiřadíme hodnotu mimo cyklus, budou náklady na takovou akci konstantní. Pokud ji však budeme přiřazovat hodnotu uvnitř cyklu, budou náklady výrazně vyšší.

2.14 Obfuskace dat

Mezi obfuskace dat patří všechny transformace, které se snaží maskovat datové struktury použité v programu. Collberg [35] je nazývá jako "rozbití abstrakcí a destrukturalizace datových struktur"*("breaking abstractions and unstructuring data structures")*. Takové transformace datových struktur můžeme podle druhu změny v kódu rozdělit do čtyř kategorí:

- Změna způsobu ukládání dat - rozdělení proměnných (např. řetězců), použití objektů namísto skalárních datových typů, globalizace lokálních proměnných, převod statických dat do funkcí, šifrování proměnných
- Změna skládání dat - skládání proměnných, změna struktury polí, změna dědičnosti tříd
- Změna pořadí dat - změna pořadí deklarací proměnných, definicí funkcí a dat v polích

2.14.1 Změna způsobu ukládání dat

Rozdělení proměnných

Proměnné, jejichž rozsah hodnot je omezený, můžou být rozdělené na více proměnných. Myšlenkou takové obfuskace je zmást revezního inženýra maskováním skutečného počtu použitých proměnných a jejich datových typů. Pokud budeme chtít rozdělit například proměnnou V typu T na dvě další proměnné p a q typu U, budeme si muset definovat následující tři typy informací [39]:

1. Funkci *f(p,q)*, která mapuje hodnoty proměnné p a q na odpovídající hodnotu V

2. Funkci *g(V)*, která mapuje hodnotu proměnné V na odpovídající hodnoty p a q

3. Operace, které budou prováděny s proměnnými p a v (např. jejich sečtení, vynásobení další konstantou, atd.)

V následujících Tabulce 2 můžeme vidět příklad rozdělení proměnné V typu boolean na dvě proměnné p a q typu integer, které nabývají hodnot 0 nebo 1. Pokud tedy v programu nastane, že $p = q = 0$ nebo $p = q = 1$, pak je hodnote V nastavena na `false`. V ostatních případech, kdy $p = 0$, $q = 1$ nebo $p = 1$, $q = 0$ je hodnota V nastavena na `true`. [36]

Vzhledem k tého nové reprezentaci proměnné typu boolean si musíme dále definovat nová zastoupení pro jednotlivé booleovské operace (AND, OR, NOT, XOR). Nejjednodušším způsobem je vytvoření tabulek pro každou operaci, které budou dostupné za běhu programu. Tyto tabulky můžou být vytvořeny obfuskátorem a uloženy jako statická data v obfuskované aplikaci, nebo generovány za běhu programu samotnou aplikací [35]. V Tabulce 3 můžeme vidět zastoupení operace AND pro proměnné A a B, jejichž hodnoty jsou vypočítány podle vzorce *2p+q* z Tabulky 2.

g(V)		*f(p,q)*	
p	q	*V*	*2p+q*
0	0	false	0
0	1	true	1
1	0	true	2
1	1	false	3

Tabulka 2: Rozdělení proměnné V na p a q (převzato z [35])

		A			
AND[A,B]		0	1	2	3
B	0	3	1	2	3
	1	1	1	2	2
	2	2	2	1	1
	3	0	1	2	0

Tabulka 3: Nové zastoupení booleovské operace AND (převzato z [35])

Účinnost a odolnost této obfuskace roste s počtem proměnných, do kterých je původní proměnná rozdělena. To však zvyšuje i náklady.

Použití objektů namísto skalárních datových typů

Skalarní datové typy (boolean, integer, float, string, atd.) mohou být transformovány do generických typů. Příkladem může být například datový typ integer, který transformujeme do třídy Integer a pokaždé, když jej chceme použít vytvoříme novou instanci této třídy. Taková třída může dále obsahovat různé metody, například pro sečtení, násobení, atd. Použití této transformace samotné nám nepřináší příliš účinnosti a odolnosti, nicméně může být mnohem efektivnější v kombinaci s dalšími transformacemi.

```
// Originální kód
int i = 0;
if(i < 3) {
   i++;
}
```

```
// Transformovaný kód
Integer i = new Integer(0);
if(i.value < 3) {
   i.value++;
}
```

Výpis 4: Obfuskace použitím objektů namísto skalárních datových typů

Globalizace lokálních proměnných

Mějme v kódu programu dvě metody *F1* a *F2*. Obě tyto metody ve svých tělech vytvářejí lokální proměnnou p typu T. Pokud se v programu nevykonávají tyto metody současně, můžeme proměnnou p typu T deklarovat jako globální a bude tak sdílená pro obě metody. Taková transformace může revezrního inženýra zmást, protože nebude tušit, že se vlastně jedná o dvě odlišné proměnné, avšak bylo využito toho, že se k nim nepřistupuje ve stejný čas z více metod najednou, a proto se globalizovaly do jedné.

```
// Originální kód
void F1() {
   int p = 0;
}
void F2() {
   int q = 2;
}
```

```
// Transformovaný kód
int P;
void F1() {
   P = 0;
}
void F2() {
   P = 1;
}
```

Výpis 5: Obfuskace globalizací lokálních proměnných

Převod statických dat do funkcí

Statická data jako například řetězce znaků obsahují spoustu informací pro revezrní inženýry. Jednoduchý způsob jak obfuskovat takový řetězec je vytvořit funkci, která jej vrací. Taková funkce může vracet i několik dalších řetězců na základě parametrů, které do ni vložíme. Není však nejvhodnější vytvářet pouze jednu funkci, která bude vracet všechna statická data daného programu, jelikož při odhalení této metody má útočník přístup ke všem datům. Mnohem vyšší účinnosti a odolnosti této obfuskace dosáhneme vytvořením více funkcí, které budou tyto data vracet, nejlépe každá jiným způsobem.

```
// Originální kód
string s1 = "ABC";
string s2 = "DEF";
string s3 = "GHI";
```

```
// Transformovaný kód
string GetString(int a, int b) {
   if(a + b == 0)
      return "ABC";
   else if (a + b == 1)
      return "DEF"
   else
      return "GHI";
}
```

Výpis 6: Obfuskace převodem statických dat do funkcí

Šifrování proměnných

Jednoduchou transformací kódu je zašifrování hodnoty jedné proměnné pomocí hodnot dalších proměnných. Příkladem může být například šifrovaní proměnné i typu integer pomocí vzorce $i = c_1 * i + c_2$, kde c_1 a c_2 jsou konstanty. V následující ukázce můžeme vidět šifrování hodnoty i v cyklu for dle tohoto vzorce kde $c_1 = 2$ a $c_2 = 3$.

```
// Originální kód
for(int i = 0; i < 50; i++)
{
   pole[i] = i;
}
```

```
// Transformovaný kód
for(int i = 3; i < 103; i++)
{
   pole[(i-3)/2] = i;
   i++;
}
```

Výpis 7: Obfuskace šifrováním proměnných

2.14.2 Změna skládání dat

Dnešní objektově orientované jazyky jsou narozdíl od strojově orientovaných jazyků mnohem více zaměřeny na ukládání dat do datových struktur než na samotné řízení běhu programu na daném počítači. Úkolem reverzního inženýrství je u objektově orientovaných aplikací především zpětné získání dat z těchto struktur. Nejčastěji používanými datovými strukturami moderních jazyků jsou objekty a pole.

Skládání proměnných

Známou metodou změny skládání dat je spojování proměnných skalárních datových typů do jedné, a to za předpokladu, že se celkový rozsah těchto proměnných vejde do rozsahu finální proměnné. Příkladem můžou být dvě proměnné A a B typu 32bit integeru, které jsou spojeny do proměnné C typu 64bit longu podle vzorce $C = (2^{32} * B + A)$. [39]
Odolnost takové transformace je poměrně nízká, protože na takovou transformaci může deobfuskátor snadno přijít zkoumáním aritmetických operací na konkrétní proměnné. Lepší variantou této transformace je složení proměnných stejného typu (avšak s rozdílnými rozsahy) do jednoho pole, které je může pojmout všechny (například složení proměnné typu short, integer a long do pole typu long). [35]

Změna struktury polí

Pole jsou důležitou datovou strukturou v programovacích jazycích. Pokud máme zmást reverzního inženýra nebo deobfuskátor, musíme zamaskovat jejich data, nebo celkově smysl jejich existence (k čemu jsou v programu použity). S poli můžeme provádět několik transformací. Nejčastěji používanými jsou rozdělení na více polí, spojení dvou a více polí do jednoho, navýšení a snížení počtu jejich dimenzí.
Na následující ukázce kódu můžeme vidět rozdělení jednoho pole na dvě. Při přiřazování hodnot v cyklu musíme podle hodnoty čítače určit, do kterého z polí se hodnota vloží.

```
// Originální kód
int[] A = new int[10];
for(int i = 0; i < 10; i++)
{
   pole[i] = i;
}
```

```
// Transformovaný kód
int[] A = new int[5];
int[] B = new int[5];
for(int i = 0; i < 10; i++)
{
   if((i%2) == 0)
      A[i/2] = i;
   else
      B[i/2] = i;
}
```

Výpis 8: Obfuskace změnou struktury polí

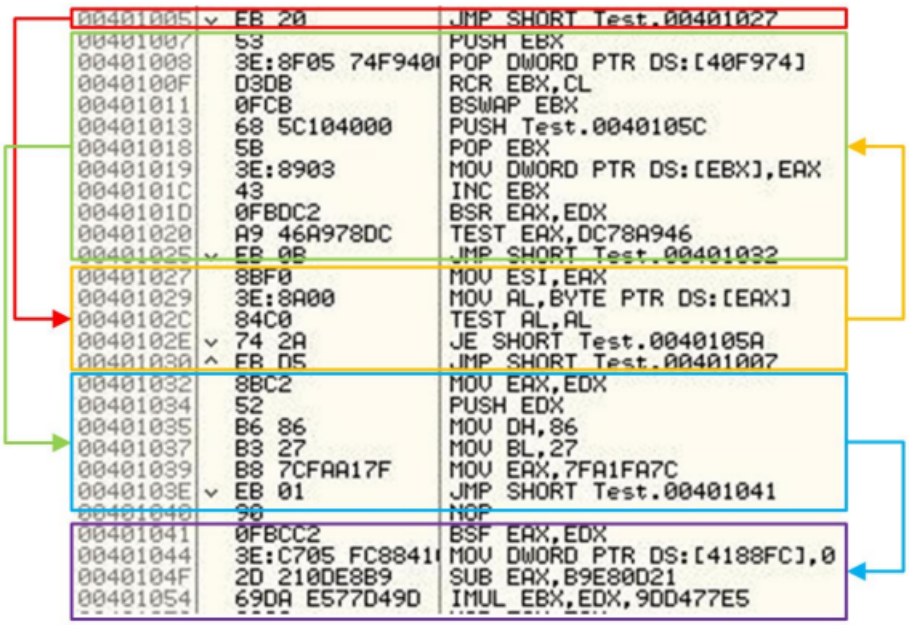
Rozdělení a navýšení dimenze polí dělá kód složitejším, takže zvyšuje účinnost obfuskace. Na druhou stranu však spojování a zmenšování počtu dimenzí snižuje složitost kódu. V tomhle případě to ale nevadí, protože kód je i tak obfuskován do podoby, která může zmást deobfuskátor i reverzního inženýra.

2.15 Obfuskace běhu programu

Mezi obfuskace běhu programu se řadí takové transformace kódu, které se snaží maskovat průběh jeho vykonávání. Typickým příkladem je změna pořadí sekvence instrukci, spojování více metod do jedné, vkládání mrtvého kódu nebo rozšiřování podmínek pro ukončení cyklu. [40][35]

2.15.1 Změna pořadí sekvence instrukcí

Změna pořadí sekvence assemblerovských instrukcí bez narušení původní funkčnost programu je jednou z nejčastěji používaných obfuskačních technik. Provádí se tak, že se určitá sekvence assemblerovských instrukcí jdoucích za sebou rozdělí na více částí, které se mezi sebou náhodně popřehází. Na konec každé části se přidá instrukce `JMP`, která přesune vykonávání kódu na tu část, která se má provést, aby byla zachována původní funkčnost programu. Na obrázku 10 můžeme vidět tuto transformaci s rozdělením kódu na 5 částí.



Obrázek 10: Ukázka změny pořadí sekvence assemblerovských instrukcí (převzato z [41])

2.15.2 Vkládání mrtvého kódu

Pod mrtvým kódem si můžeme představit takový kód, který se sice provede, ale nijak nenaruší původní funkčnost programu. Jedná se tedy o přebytečný kód, jehož vložením chceme zmást antivirový program nebo reverzního inženýra. V assemblerovském kódu může jít například o vkládání instrukcí `NOP`, které nic neprováději. V dnešní době však spousta antivirových programů dokáže automaticky odstranit přebytečné instrukce `NOP`. Dalším příkladem mrtvého kódu je například inkrementace a následná dekrementace číselné proměnné o stejnou hodnotu, nebo podmínky, které budou vždy pravdivé. Na obrázku 11 můžeme vidět vkládání mrtvého kódu v assembleru.

Obrázek 11: Ukázka obfuskace vkládáním mrtvého kódu v assembleru (převzato z [41])

2.15.3 Rozšiřování podmínek pro ukončení cyklů

Podmínky pro ukončení cyklů jsou dobrým místem pro aplikaci obfuskace. Můžeme je rozšířit o další jednu nebo dvě podmínky, které nijak nenaruší průběh a počet opakování cyklu.

```
// Originální kód
int i = 1
while(i<50)
{
   ...
   i++;
}
```

```
// Transformovaný kód
int i = 1
int j = 20
while(i < 50 && j % 2 == 0)
{
   ...
   i++;
   j+2;
}
```

Výpis 9: Obfuskace rozšiřováním podmínek pro ukončení cyklů

2.16 Obfuskace struktury programu

Obfuskace struktury programu patří mezi nejjednodušší. Jedná se o takové transformace zdrojového kódu, které mění jeho strukturu. Typickým příkadem je například odstranění komentářů, změna formátování nebo přejmenování proměnných a metod. Takové transformace jsou však jednosměrné a reverzní inženýr již nikdy nebude schopen dostat zpátky původní kód.

3 Praktická část

Cílem praktické části je vytvoření analyzační aplikace pro spustitelné PE soubory (EXE). Kromě tvorby analyzační aplikace je provedeno také několik experimentů s packery představenými v teoretické části a antivirovými programy.

3.1 Specifikace analyzační aplikace

Aplikace má sloužit k analýze souborů PE formátu, konkrétně formátu EXE. K implementaci byla vybrána platforma .NET s programovacím jazykem C#. Jedná se o formu konzolové aplikace, která zpracovává soubory jednotlivě. Pro načtení souboru do aplikace je nutno použít příkazové řádky a danými parametry zadat cestu k souboru, který chceme analyzovat.

Scénář použití analyzační aplikace je následující:

1. Uživatel z příkazové řádky parametrem `-f` zadá cestu k souboru, který chce analyzovat
2. Aplikace se snaží zjistit, zda byl na zkoumaný soubor použit packer, či nikoliv
3. Pomocí volitelného parametru `-ml` je spuštěna detekce použitého packeru

Jednotlivé analyzační metody, se kterými aplikace pracuje jsou popsány v následujících kapitolách.

3.2 Použité knihovny a nástroje

3.2.1 PeNet

PeNet[42] je knihovna pro snadné parsování 32bit a 64bit PE souborů pro platformu .NET. S parsovaným PE souborem se díky této knihovně lépe pracuje, jelikož objekt třídy `PeFile`, jež představuje parsovaný soubor, obsahuje veškeré informace o hlavičkách, sekcích, importovaných funkcích z Windows API a další data o stuktuře PE souboru, která vychází na základě oficiální dokumentace PE souborů od společnosti Microsoft. Jelikož je knihovna ve vývoji teprve od roku 2018 a stále se vyvíjí, je možné, že uživatel nedostane všech informací o PE souboru, které by očekával. Pro experimentální a výzkumné účely je více než dobrá a užitečná. Knihovnu lze stáhnout formou NuGet balíčku do projektu ve Visual Studiu.

3.2.2 ML.NET

ML.NET[43] je open-source multiplatformní framework pro aplikaci strojového učení. Je vyvíjen společností Microsoft od roku 2018 a funguje na platformě .NET a .NET Core. Velkou výhodou tohoto frameworku je, že uživatel nemusí mít předem rozsáhlé znalosti o vytváření a fungování modelů strojového učení, a proto je určena i začátečníkům v této oblasti. V analyzační aplikaci je framework ML.NET použit k binární klasifikaci komprimovaného souboru a vícetřídní klasifikaci

(*multi-class clasification*) packeru použitého na zkoumaný soubor. Model je trénován datovým setem, jehož vytváření je popsáno v kapitole 3.3.

3.2.3 OllyDbg

OllyDbg je disassembler a debugger pro platformu Windows. Díky breakpointům, které si můžeme nastavit, jsme schopni zkoumanou aplikaci debugovat a analyzovat její chování. V praktické části je OllyDbg použit pro manuální dekompresi společně s pluginem OllyDump, který z aktuálního procesu v paměti vytvoří nový PE soubor.

3.2.4 ImpREC

Nástroj ImpREC slouží k obnovení tabulky importů komprimovaných souborů. Některé automatické unpackery soubor sice dekomprimují, ale neobnoví tabulku importů, a tak program nelze spustit. ImpREC po připojení na aktivní proces programu debugovaného v OllyDbg, a po zadání OEP, je schopen nalézt používané funkce Windows API a obnovit tak tabulku importů.

3.3 Příprava pracovních PE souborů

Dříve než budou popsány konkrétní metody použité v analyzační aplikaci, je potřeba zajistit větší množství nekomprimovaných PE souborů, které budou komprimovány různými packery a se kterými se bude následně pracovat. Tyto soubory slouží především k vytváření datových setů pro metody založené na strojovém učení, jež jsou popsány v následujících kapitolách.
Jako důvěrný zdroj nekomprimovaných PE souborů byla vybrána čistá instalace 32bit verze Windows 10 Home. Důvod, proč byla vybrána pouze 32bit verze je ten, že většina použitých packerů pracuje pouze s 32bit PE soubory, a proto nebyly 64bit soubory vůbec použity. Cílovými soubory byly všechny soubory EXE obsažené v systémovém adresáři `C:\Windows` (včetně podadresářů). Ke snadné extrakci všech těchto souborů byl vytvořen dávkový soubor, který je překopíruje do námi zvoleného adresáře.

```
set source="C:\Windows"
set destination="C:\windows_pe_all"
for %%F in (%destination%) do set destination="%%~fF"
for /r %source% %%F in (.) do if "%%~fF" neq %destination%
ROBOCOPY "%%F" %destination% *.exe /COPYALL /R:0
```

Výpis 10: Dávkový soubor pro extrakci všech EXE souborů z adresáře C:\Windows

Celkově bylo z adresáře `C:\Windows` vyextrahováno 848 EXE souborů, z nichž byly v počtu 123 vyřazeny ty, které fungují na platformě .NET (mají jinou strukturu a 4 z 5 použitých packerů v této studii s nimi nejsou kompatibilní). Po vyřazení zbylo tedy celkem **725 EXE souborů** korektního formátu PE32.

V poslední fázi přípravy bylo použito celkem 5 packerů popsaných v teoretické části - UPX (ver. 3.95) ASPack (ver. 2.43), PECompact (ver. 3.0.2), PELock (ver. 2.09) a Obsidium (ver. 1.6.7) ke kompresi vyextrahovaných souborů. Každý packer byl použit na všech 725 souborů. U některých packerů docházelo k menší chybovosti při kompresi, a tak byly tyto soubory (pro daný packer) vynechány. Celkově bylo dosaženo počtu **3566 úspěšně komprimovaných souborů**.

3.4 Detekce komprimovaného souboru

V prvé řadě se analyzační aplikace snaží zjistit, zda na zkoumaný soubor byl použit packer, či nikoliv. V této fázi se však zatím nebere ohled na to, jaký packer byl použit. Jedná se o pouhou klasifikaci komprimovaného a nekomprimovaného souboru.
Abychom mohli určit, zda na daný soubor byl použit packer, je potřeba si nejprve definovat způsob klasifikace, podle kterých bude soubor spadat mezi komprimované či nekomprimované. Následující techniky klasifikace jsou inspirovány již existující studií *"Classification of Packed Executables for Accurate Computer Virus Detection"*[46] z roku 2008.

3.4.1 Podmínková klasifikace

Tento způsob klasifikace komprimovaného souboru zatím nevyužívá algoritmů strojového učení, nýbrž je založen na prostých podmínkách pro různé charakteristiky zkoumaného PE souboru. Tyto charakteristiky jsou především takové vlastnosti PE souboru, podle kterých lze snadno zjistit, že je soubor komprimovaný (viz kapitola 2.9.1 o obecných známkách komprimovaných souborů).
V této metodě je definováno celkem **9 charakteristik PE souboru**, podle jejichž hodnot je soubor klasifikován jako komprimovaný či nekomprimovaný:

1. **Počet nestandartních sekcí** - Jak již bylo zmíněno v teoretické části, komprimované soubory často obsahují sekce s nestandartními názvy. Abychom mohli definovat jaký název sekce je nestandartní, musíme si nejprve určit názvy standartní. K získání seznamu standartních sekcí byly extrahovány všechny EXE soubory z adresáře Windows operačního systému Windows 10. Z těchto souborů byly následně extrahovány názvy jejich sekcí. Seznam lze nalézt v příloze A. Pokud nějaká ze sekcí obsažených ve zkoumaném souboru není v tomto seznamu, je započítána jako nestandartní. Bylo zjištěno, že u komprimovaných souborů se často vyskytují i názvy vytvořené pouze z mezer, takže se tváří jako bez názvu.

2. **Počet standartních sekcí** - Standartní sekce se opět určují podle seznamu v příloze A. U komprimovaných souborů se může dokonce stát, že neobsahují ani jednu standartní sekci. Ve většině případech mají alespoň sekci `.rsrc` se soubory jako obrázky, ikony, zvuky apod. Spousta packerů však nabízí možnost volby komprese či šifrování i této sekce.

3. **Počet sekcí s povoleným vykonáváním kódu** (*executable sections*) - V běžném PE souboru bývá pouze jedna sekce s povoleným vykonáváním kódu, a sice sekce `.text`. Pokud je počet takových sekcí větší, s největší pravděpodobností je soubor komprimován. Packery tyto přídavné sekce vytvářejí zejména jako místo pro zápis dekomprimovaného a dešifrovaného kódu, který je následně vykonáván. Sekce s povoleným vykonáváním kódu jsou označeny flagem `IMAGE_SCN_MEM_EXECUTE`.

4. **Počet sekcí s povoleným vykonáváním kódu, čtením a zápisem** - Jedná se o podobný případ jako v předchozím bodě, avšak s tím rozdílem, že tyto sekce mají navíc povolené čtení a zápis. V běžném PE souboru je to opět pouze sekce `.text`. Většina packerů však přídavné sekce vytváří se všemi právy (čtení, zápis, vykonávání kódu), a tak je počet sekcí v tomto bodě často shodný s počtem sekcí v bodě předchozím. Tyto sekce jsou označeny flagy `IMAGE_SCN_MEM_EXECUTE`, `IMAGE_SCN_MEM_READ` a `IMAGE_SCN_MEM_WRITE`.

5. **Počet importovaných funkcí Windows API** - Komprimované PE soubory mívají daleko méně importovaných funkcí Windows API než nekomprimované. Po zkoumání souborů komprimovaných pěti packery (viz kapitola 2.8) je počet těchto importů v průměru menší než 10.

6. **Entropie PE hlavičky** - Jak již bylo zmíněno v kapitole 2.9.2, data komprimovaných souborů mají díky své nahodilosti vysokou bajtovou entropii. Ačkoliv packery vkládávají komprimovaná a šifrovaná data většinou do sekcí, i v PE hlavičce jsou části, do kterých můžeme vkládat data, aniž by byla narušeno správné načtení programu do paměti při spuštění. Jedná se především o část nepovinné hlavičky (*optional header*).

7. **Entropie sekcí se spustitelným kódem** - Měření entropie sekcí se považuje za jednu z nejdůležitějších částí klasifikace komprimovaných souborů, jelikož právě jejich obsah bývá nejčastěji komprimován, a to především sekce se spustitelným kódem. Za sekce se spustitelnm kódem můžeme považovat všechny sekce v souboru, jež jsou označeny flagem `IMAGE_SCN_MEM_EXECUTE`. Pokud je těchto sekcí více, pro účely měření jsou jejich data spojena do jednoho bufferu, nad kterým je výpočet entropie proveden.

8. **Entropie sekcí s daty** - Některé packery nabízejí možnost komprese sekcí s daty aplikace (např. sekce `.rsrc`). Pokud je taková sekce komprimována, entropie roste také. Za sekce z daty jsou ve zkoumaném souboru považovány všechny sekce, které nejsou označeny flagem `IMAGE_SCN_MEM_EXECUTE`

9. **Entropie celého souboru** - Je měřena na bufferu celého souboru bez ohledu na sekce nebo hlavičky.

Na základě výše uvedených charakteristik vznikla tabulka č. 4 s rozsahy hodnot pro jednotlivé charakteristiky. Jedná se o rozšíření tabulky ze studie [46] o hodnoty, pro které je zkoumaný

soubor klasifikován jako komprimovaný. Na základě těchto hodnot analyzační aplikace určuje, zda je soubor komprimován, či nikoliv (viz obrázek 12)

Tabulka 4: Rozsahy vybraných charakteristik PE souboru a hodnoty pro klasifikování souboru jako komprimovaný

Charakteristika	**Rozsah hodnot**	**Hodnoty pro kompresi**
Počet nestandartních sekcí	integer ≥ 0	≥ 1
Počet standartních sekcí	integer ≥ 0	< 2
Počet sekcí s povoleným vykonáváním kódu	integer ≥ 0	> 1
Počet sekcí s povoleným vykonáváním kódu, čtením a zápisem	integer ≥ 0	> 1
Počet importovaných funkcí Windows API	integer ≥ 0	< 10
Entropie PE hlavičky	$[0, 8]$	≥ 7
Entropie sekcí se spustitelným kódem	$[0, 8]$	≥ 7
Entropie sekcí s daty	$[0, 8]$	≥ 7
Entropie celého souboru	$[0, 8]$	≥ 7

CHARACTERISTIC	VALUE	PREDICTION
Nr. of standart sections	2	OK
Nr. of non-standart sections	2	PACKED
Nr. of executable sections	2	PACKED
Nr. of R, W, E sections	2	PACKED
Nr. of Windows API imports	6	PACKED
Entropy of header	2,17789	OK
Entropy of code sections	7,83584	PACKED
Entropy of data sections	3,4755	OK
Entropy of entire file	7,28145	PACKED

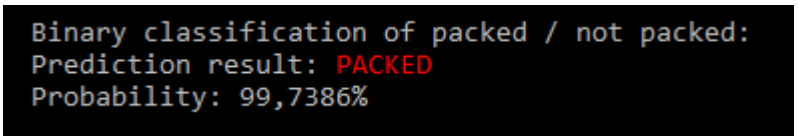
Obrázek 12: Ukázka podmínkové klasifikace v analyzační aplikaci (soubor byl komprimován packerem)

3.4.2 Binární klasifikace

Cílem tohoto řešení je vyzkoušet použití strojového učení ke klasifikaci zkoumaného souboru jako komprimovaný či nekomprimovaný. Implementace je v analyzační aplikaci provedena za pomocí frameworku ML.NET[43]

V první fázi je potřeba vytvoření datového setu pro následné trénování budoucího modelu. K tomu jsou použity vyextrahované soubory zmíněné v kapitole 3.3. Jsou vybrány všechny nekomprimované soubory v původním stavu (725) a dalších 725 náhodně vybraných komprimovaných souborů z celkového počtu 3574. Celkem tedy vznikne datový set o 1450 záznamech. Pro

vytvoření jednotlivých záznamů je potřeba si nejprve definovat tzv. vektor vlastností (*feature vector*) a značky (*labels*). Za vlastnosti (*features*) je vybráno všech 9 charakteristik již definovaných v předchozí kapitole o podmínkové klasifikaci. Za značky jsou v tomto případě považovány příznaky `true` a `false` (soubor buď je nebo není komprimován). Jeden záznam datového setu bude mít tedy celkem 10 hodnot (9 charakteristik PE souboru + značka).
V další fázi následuje trénování a evaluace modelu. Jako klasifikační algoritmus je zvolen **rozhodovací strom**. K vyhodnocení úspěšnosti modelu je využito **křížové validace**, při které je datový set rozdělen na 70% trénovací a 30% testovací set. Klasifikátor nejprve natrénuje model na trénovací množině a následně jej testuje na testovací množině. Celý tento proces je opakován 10 krát a pokaždé s jinou trénovací a testovací množinou. Pro tento datový set je **dosaženo úspešnosti 99,92%**.
V dalším pokusu je využito stejných postupů, avšak v datovém setu jsou obsaženy všechny komprimované soubory (3566) a všechny nekomprimované (725). Celkem je tedy datový set obsažen 4291 záznamy. V takovém případě je dosaženo úspěšnosti 99,89%.
Analyzační aplikace využívá tohoto modelu ke klasifikaci zkoumaného souboru jako komprimovaný či nekomprimovaný. Výstupem je příznak `PACKED` nebo `OK`, a pravděpodobnost, s jakou do dané třídy patří (viz obrázek 13).



Obrázek 13: Ukázka binární klasifikace v analyzační aplikaci (soubor byl komprimován packerem)

Ačkoliv se model dle křížené validace zdá při obou datasetech úspešný, nemusí tomu tak být i při použití jiného testovacího datového setu. Pro další ověření úspešnosti modelu bylo navíc staženo celkem 20 EXE souborů ze serveru Slunečnice.cz a 10 z nich bylo následně komprimováno. Při analýze těchto souborů byly všechny soubory rozeznány správně jako komprimované a nekomprimované, ovšem pravděpodobnosti, že je soubor komprimovaný se nejčastěji pohybovaly kolem 99-100% a pravděpodobnosti, že je nekomprimovaný byly často v rozmezí 10-30%. Přesný důvod, proč jsou tyto pravděpodobnosti v takových rozdílech nebyl zjištěn, ovšem jedna z možností je nedostatečně velký datový set, nebo naopak příliš velký datový set. Studie[46], ze které tento experiment vychází a dosahuje lepších výsledků, pracuje s mnohem větším a různorodějším datovým setem. Jejich datový set se skládá s více než 2000 nekomprimovaných souborů z více zdrojů, než je pouze OS Windows, a více než 2500 komprimovaných virů.

3.5 Detekce použitého packeru

Cílem tohoto řešení je detekovat použitý packer na zkoumaný soubor (pokud byl detekován jako komprimovaný). K implementaci je použito opět algoritmů strojového učení, ovšem v tomto případě se jedná o vícetřídní klasifikaci (*multi-class classification*), jelikož klasifikujeme soubor do více tříd (podle názvu packeru).
Tvorba datového setu probíhá podobně jako při binární klasifikaci. Za vektor vlastností je opět zvoleno 9 charakteristik PE souboru popsaných v kapitole 3.4.1. Značky budou tentokrát představovat název packeru. Pro extrakci dat je zvoleno všech 3566 souborů komprimovaných celkem 5 packery (UPX, ASPack, PECompact, PELock, Obsidium).
Pro trénování modelu jsou vybrány klasifikátory Stochastic Dual Coordinate Ascent (SDCA) a Naive Bayes (současně framework ML.NET nabízí pouze tyto dva klasifikátory k použití u vícetřídní klasifikace). Evaluace modelu je provedena opět pomocí křížené validace s počtem opakování 10 rozdělením datového setu na 70% trénovací a 30% testovací. Úspěšnost obou klasifikátorů lze shlédnout v následující tabulce č. 5.

Tabulka 5: Úspěšnost modelu pro detekci packeru

Klasifikátor	**Úspěšnost (křížená validace)**
SDCA	99,79%
Naive Bayes	59,43%

Tento způsob detekce packeru pomocí strojového učení je však omezen pouze na packery obsažené v datovém setu. Další problém nastává, pokud je soubor komprimován neznámým packerem. V takovém případě je tato metoda nepoužitelná.

3.6 Dekomprese

Původním plánem bylo zakomponovat dekompresi do analyzační aplikace. K tomu je potřeba analyzovaný soubor disassemblovat, emulovat jeho spuštění, během kterého se nalezne OEP, a udělat dump dekomprimovaného kódu. Bylo tedy vyhledáváno řešení pro platformu .NET, které by tento proces emulace provedlo. Jediná knihovna, která dokáže emulovat assemblerovský kód v .NET aplikaci je ASM.Net[20]. Ovšem bylo zjištěno, že tato knihovna nedokáže PE soubor disassemblovat a vytvářet breakpointy v určitých místech kódu (pro nalezení OEP), a proto byl tento plán zavrhnut. Existují však jiná řešení a návody pro tvorbu vlastního unpackeru (např. [21]), avšak ty jsou psány v C++, které je pro řešení této problematiky mnohem lepší než .NET. Namísto implementace vlastního unpackeru je tedy provedena dekomprese automatická, pomocí nástrojů představených v teoretické části (viz kapitola 2.12), a manuální, pomocí nástroje OllyDbg.

3.6.1 Automatická

Pro vyzkoušení automatické dekomprese je nejprve stažen náhodný EXE soubor z portálu Slunečnice.cz, který splňuje formát PE32. Pro zjednodušení je automatická dekomprese zaměřena pouze na packer AsPack (ver. 2.43), kterým je soubor komprimován. Použity jsou unpackery **AspackDie**, **FUU (AsPack plugin)** a **PEiD Generic Unpacker** (PEiD GU). Po dekompresi jsou pozorovány různé vlasnosti dekomprimovaného souboru, jako např. zda jej lze spustit, zda byla obnovena tabulka importů, nebo entropie celého souboru. Dále jsou z něj pomocí nástroje **Strings**[44] vyextrahovány všechny řetězce s minimálním počtem znaků 6 a je sečten počet těch, které jsou smysluplné (v kontextu PE souboru dávají smysl a nejsou to náhodné sekvence znaků). Výsledky dekomprese lze videt v tabulce 6.

Tabulka 6: Výsledky automatické dekomprese komprimovaného souboru packerem AsPack pomocí 3 unpackerů (včetně porovnání s originálním a komprimovaným souborem)

	Originální	Komprimovaný	AspackDie	FUU	PeID GU
OEP nalezen	-	-	✓	✓	✓
Aplikaci lze spustit	✓	✓	✗	✓	✗
Importy obnoveny	-	-	✗	✓	✗
Resources obnoveny	-	-	✓	✓	✓
Entropie souboru	4,82	7,15	3,53	5,53	4,36
Smysluplných řetězců	126	45	45	220	146
Velikost souboru	36 KB	21 KB	52 KB	37 KB	56 KB

Na výsledcích automatické dekomprese lze vidět, že nejlepší práci odvedl unpacker FUU s pluginem pro packer AsPack. Jako jedinému se mu povedlo obnovit tabulku importů a dekomprimovanou aplikaci bylo možné úspěšně spustit. U unpackerů AspackDie a PEiD GU bohužel nebyla obnovena tabulka importů, což i zapříčinilo to, že aplikace nebyla spustitelná. Zajímavě také dopadla analýza extrahovaných řetězců. U unpackeru AspackDie se jejich počet od komprimovaného souboru nijak nezvýšil a jejich seznam je naprosto totožný. Tento fakt způsobuje především absence tabulky importů, ve které jsou obsaženy názvy funkcí Windows API, které se mezi tyto extrahované řetězce započítávají. U souboru dekomprimovaného unpackerem PEiD GU bylo zjištěno, že názvy importovaných funkcí jsou sice v souboru obsaženy, ale v tabulce importů nejsou. Je tedy jasné, že unpacker tyto funkce do tabulky importů vůbec nezapsal. Z hlediska entropie souborů si lze povšimnou, že po dekompresi se její hodnota snížila, což potvrzuje fakt, že komprimované soubory mají entropii značně vyšší, než soubory originální. Velikosti některých dekomprimovaných souborů jsou značně vyšší, než u originálního souboru. Je to způsobeno tím, že spousta unpackerů po dekompresi zanechá v souboru spoustu nepotřebných dat (komprimovaný kód, nulové bajty, nepotřebné sekce). V tomto případě nejlépe dopadl opět unpacker FUU, který tyto nepotřebná data v dekomprimovaném souboru odstranil.

Automatická dekomprese je jednoduchá, protože se veškerou práci za nás snaží udělat unpacker a uživatel tak nemusí mít větší znalost assembleru. Ovšem sehnat unpacker, který soubor dekomprimuje a zároveň obnoví tabulku importů, může být složité. Problémy nastavají například při příchodu nových verzí packerů, které můžou přinést změny v dekompresním algoritmu a unpacker si s nimi nemusí poradit. V takovém případě je potřeba unpacker aktualizovat společně s packerem. Dalším problémem jsou generické unpackery, na které se nelze úplně spolehnout, jelikož pro některé komprimované soubory fungují, a pro jiné zase ne. Tvorba kvalitního generického unpackeru je stále budoucností a dosti možná pomocí algoritmů umělé inteligence. Ve značné výhodě s automatickou dekompresí jsou antivirové společnosti, kterým vývojáři některých packerů předkládají přesné postupy k dekompresi svých packerů, a tak nemusí používat ty, které jdou běžně stáhnout z internetu.

3.6.2 Manuální

Pro manuální dekompresi je vybrán stejný soubor jako při dekompresi automatické, avšak tentokrát je komprimován packerem **UPX**. Dekomprese je provedena podle návodu pro manuální dekompresi packeru UPX nalezeného na internetu[22]. Pro debugování je použit nástroj OllyDbg (ver 1.10) a pro obnovení tabulky importů ImpREC (ver. 1.7e).
V první fázi manuální dekomprese je komprimovaný soubor nahrán do debuggeru OllyDbg. Následně je potřeba nalézt OEP. Toho docílíme tak, že nastavíme breakpoint na zásobník (registr `ESP`), který vykonávání kódu zastaví na konci dekompresního algoritmu UPX. Zde již můžeme vidět, že poslední instrukce dekompresního algoritmu je `JMP` na velmi vzdálenou adresu. Právě pro provedení této instrukce je vykonávání kódu přesunuto na OEP, od kterého následuje dekomprimovaný kód, který je potřeba uložit (udělat tzv. dump) jako nový EXE soubor.
V druhé fázi je zapotřebí v nově vytvořeném EXE souboru obnovit tabulku importů. K tomu je použit nástroj ImpREC, který se připojí na aktuálně spuštěný proces aplikace, kterou se snažíme dekomprimovat. Následně se do ImpREC zadá adresa OEP, která byla získána pomocí OllyDbg, a tabulka importů je obnovena automaticky. Nakonec je soubor s obnovenými importy znovu uložen.
Po manuální dekompresi packeru UPX si lze všimnout, že názvy sekcí zůstaly stejné jako když byl soubor komprimovaný. Rozdíl je však ve velikosti sekce `UPX0`, která se z nulové velikosti navýšila o velikost dekomprimovaného kódu. Také sekce `UPX1` má stále stejnou velikost, což značí, že komprimovaný kód v souboru stále zůstal, i přestože se již nedekomprimuje a nespouští, jelikož EP je nově nastaven na kód dekomprimovaný. Tím, že komprimovaný kód stále zůstal souboru také způsobuje to, že dekomprimovaný má téměř dvojnásobnou velikost. Sekce `.rsrc` zůstala stejná, jelikož packer UPX resources nedekomprimuje. Všechny výsledky manuální dekomprese lze shlédnout v tabulce 7.

Tabulka 7: Výsledky manuální dekomprese komprimovaného souboru packerem UPX (včetně porovnání s originálním a komprimovaným souborem)

	Originální	Komprimovaný	Dekomprimovaný
OEP nalezen	-	-	✓
Aplikaci lze spustit	✓	✓	✓
Importy obnoveny	-	-	✓
Resources obnoveny	-	-	✓
Entropie souboru	4,82	7,28	4,35
Smysluplných řetězců	126	42	214
Velikost souboru	36 KB	17 KB	60 KB

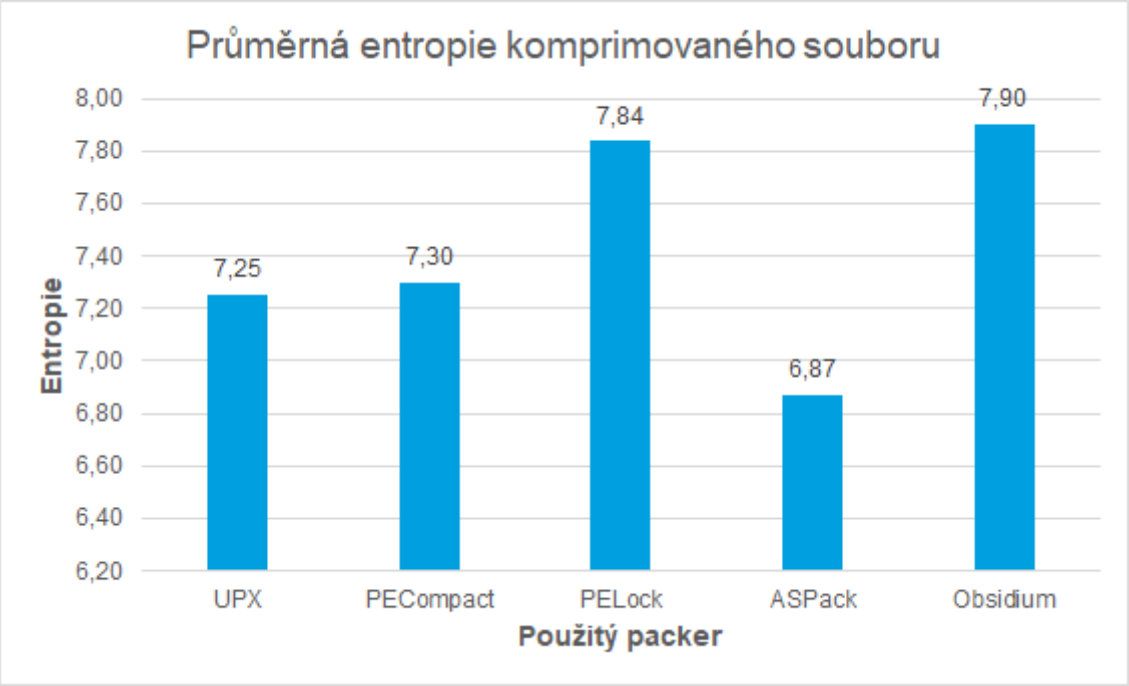
Manuální dekomprese se dá považovat za nejspolehlivější způsob jak dekomprimovat známé i neznámé packery. K jejímu provedení je však potřeba dobrá znalost assembleru a umět dobře ovládat nástroj pro disassemblování a debugování. Na internetu lze dnes nalézt spoustu návodů pro manuální dekompresi nejrůznějších packerů. Nejvíce jich je na serveru Tuts 4 You[23].

3.7 Další experimenty a pozorování

3.7.1 Pozorování entropie po aplikaci komprese

Entropie PE souboru je důležitou vlastností pro antivirové programy a ostatní analyzátory, které se snaží zjistit, zda byl na zkoumaný soubor použit packer, či nikoliv. Čím je entropie vyšší, tím je větší pravděpodobnost použití packeru. Cílem tohoto pozorování je zjistit jakou má komprese provedená packery vliv na entropii celého souboru. K pozorování byla vypočtena entropie všech 3566 komprimovaných souborů připravených v kapitole 3.3 a zprůměrována pro všech 5 packerů. Kromě toho byla také vypočtena průměrná entropie všech 725 originálních souborů, která dosahuje **hodnoty 6,15**.

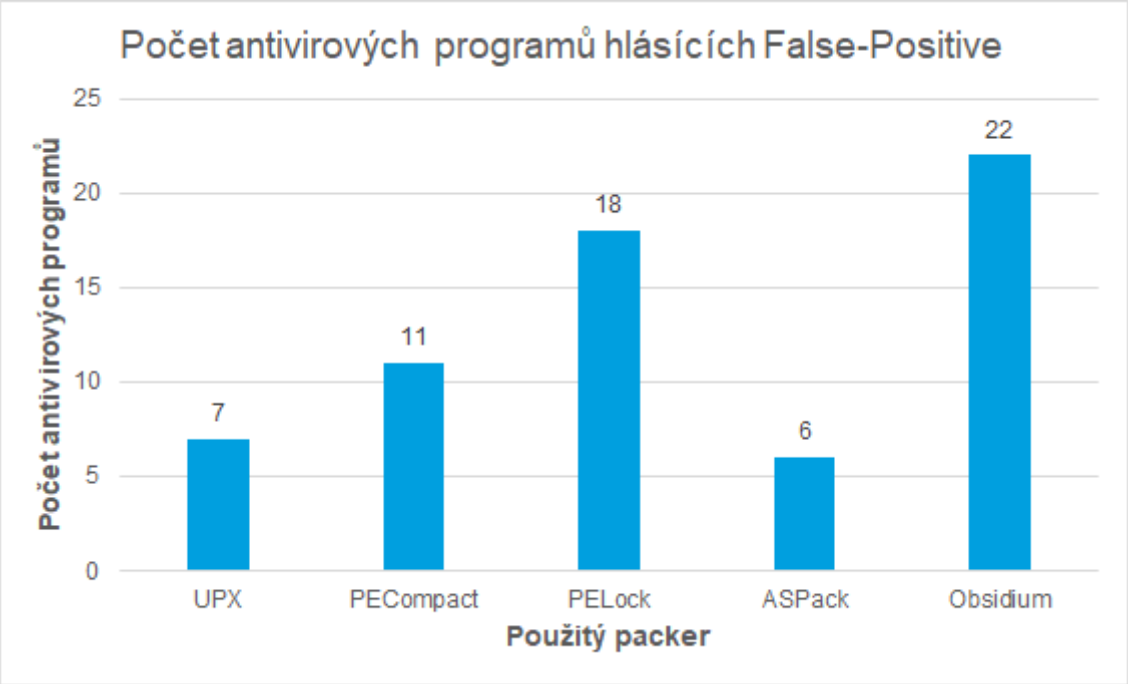
Jak lze vidět na obrázku 14, nejmenší průměrná entropie byla zaznamenána u packeru ASPack, která je už na hranici, kdy některé analyzátory detekují soubor s touto entropií jako nekomprimovaný. Dalo by se tedy říct, že z hlediska utajení malwaru by byl tento packer nejvhodnější. Nejhůře na tom však jsou packery PELock a Obsidium, u kterých se průměrná entropie pomalu blíží k maximální hodnotě 8. Jak je známo, tyto packery se řadí mezi protektory, které nabízejí mnohem více možností ochrany softwaru, než jen klasickou kompresi. I ve výchozím nastavení těchto protektorů, které bylo aplikováno ke kompresi připravených souborů, byly nastaveny ochrany jako např. rozšířená ochrana importů nebo šifrování resources (sekce `.rsrc`). Běžné packery jako UPX, PECompact a ASPack tyto ochrany nenabízí, a to je také důvod, proč protektory vykazují o něco vyšší entropii.



Obrázek 14: Průměrná entropie komprimovaných souborů dle packeru

3.7.2 Experiment s VirusTotal.com

Cílem tohoto jednoduchého experimentu je zjistit, jak na jednotlivé packery reagují antivirové programy na serveru VirusTotal.com, aniž by byl komprimovaný soubor virus. K experimentu je vybrán náhodný soubor z připravených souborů v kapitole 3.3 komprimovaný 5 packery (konkrétně aplikace Kalkulačka calc.exe). Nejprve je analyzován originální soubor, u kterého žádný z antivirů nehlásí detekci viru. Následně jsou analyzovány soubory komprimované. Pokud některý z antivirů hlásí, že zkoumaný soubor je virus, výsledek je tedy FP (False-Positive), protože soubor virem není. Kompletní výsledky pro jednotlivé packery lze vidět na obrázku 15.
Na výsledcích můžeme vidět, že nejméně byl detekován soubor komprimovaný packery ASPack a UPX. Na druhou stranu u protektorů PELock a Obsidium je falešná detekce viru značně vyšší, což může být způsobeno tím, že protektory i v základním nastavení na soubor aplikují více ochran, které jsou často aplikovány i na viry (stejný případ jako při pozorování entropie v kapitole 3.7.1).



Obrázek 15: Počet antivirových programů na serveru VirusTotal.com hlásících False-Positive

3.7.3 Experiment s vícenásobnou kompresí

Cílem tohoto experimentu je vyzkoušet vícenásobnou kompresi použitím více packerů na jeden EXE soubor reálného vzorku malwaru a pozorovat, jak se mění detekce škodlivého payloadu antivirovými programy na serveru VirusTotal.com s přibývajícími vrstvami komprese. Zvoleným vzorkem pro experiment je známý **ransomware WannaCry**, který nejvíce napadal počítace v roce 2017. Vzorek je stažen z nalezené databáze vzorků na internetu[47] a jeho hash je `32f24601153be0885f11d62e0a8a2f0280a2034fc981d8184180c5d3b1b9e8cf`. Jelikož nevíme, zda je tento vzorek již komprimován, je nejprve analyzován známými nástroji pro detekci packeru. ExeInfo PE, PEiD ani DiE nehlásí nalezený packer. Použita je tedy analyzační aplikace a metoda podmínkové klasifikace (výsledky viz obrázek 16)

CHARACTERISTIC	VALUE	PREDICTION
Nr. of standart sections	4	OK
Nr. of non-standart sections	0	OK
Nr. of executable sections	1	OK
Nr. of R, W, E sections	0	OK
Nr. of Windows API imports	91	OK
Entropy of header	0.72685	OK
Entropy of code sections	6.13459	OK
Entropy of data sections	7.97218	PACKED
Entropy of entire file	7.96426	PACKED

Obrázek 16: Analýza ransomware WannaCry analyzační aplikací

Podle analyzační aplikace v souboru nejsou žádné nestandartní sekce, počet importovaných funkcí je v běžném počtu a ani zde není více jak 1 sekce se spustitelným kódem. Podle entropie souboru, která dosahuje skoro maximální hodnoty, je soubor komprimován. Vypadá to, že autor ransomware WannaCry použil vlastní kompresní algoritmus nebo packer, který není běžně známý.
Předpokládejme, že původní vzorek je komprimovaný. V tom případě již při aplikaci prvního packeru vznikne vícenásobná komprese. Prvně je vzorek komprimován packerem PELock a napodruhé packerem Obsidium. Soubor je postupně s jednotlivými vrstvami komprese analyzován pomocí VirusTotal.com a sledovány jsou detekce známých antivirových programu jako **ESET NOD32**, **Avast**, **AVG**, **Avira**, **McAfee** a **Windows Deffender**. Kompletní výsledky lze vidět v tabulce 8.

Tabulka 8: Výsledky detekcí ransomware WannaCry pro známé antivirové programy při aplikaci vícenásobné komprese (✓= vzorek je detekován jako virus, ✗= vzorek není detekován jako virus)

	Původní	PELock	PELock+Obsidium
ESET NOD32	✓	✗	✗
Avast	✓	✗	✗
AVG	✓	✗	✗
Avira	✓	✗	✗
McAfee	✓	✗	✗
Windows Deffender	✓	✓	✗

Na výsledcích lze vidět, že nejlépe s detekcí ransomware WannaCry dopadl Windows Deffender. Všechny ostatní antiviry selhaly už při aplikaci prvního packeru PELock. Po aplikaci obou packerů virus přehlídly všechny antivirové programy.
Aplikace vícenásobné komprese či obfuskace je velmi často využívána autory malwaru. Běžně dostupné packery na internetu však nejsou na vícenásobnou kompresi příliš stavené. Například packer AsPack si soubor chrání tak, aby na něj nešel použít další packer. To, že v tomto experimentu fungovalo použití PELock a Obsidium dohromady na jeden soubor ještě neznamená, že bude spustitelný. Ovšem payload ransomware WannaCry byl těmito packery úspěšně skryt. Pro spolehlivou vícenásobnou kompresi je mnohem lepší si vytvořit packer nebo algoritmus vlastní namísto použití kombinace packerů běžně dostupných, které mezi sebou nemusí být kompatibilní.

4 Závěr

Cílem této diplomové práce bylo zaměření se na kompresní a obfuskační metoty používané u malware na platformě Windows a na metody pro jejich detekci.
V teoretické části byly nejprve představeny metody komprese, šifrování a obfuskace. Tyto metody jsou v současnosti pouzívány nejen hackery k maskování těl škodlivých programů, ale i k ochraně užitečného software před kráděží know-how. Byla popsána struktura PE souboru na platformě Windows, kterou je nutno znát pro případnou statickou i dynamickou analýzu. Packery jsou v dnešní době nejpoužívanějšími nástroji pro maskování softwaru a jejich aplikace se nazývá komprese spustitelných souboru nebo anglicky *packing*. Kromě kompresního algoritmu, který zmenší velikost spustitelného souboru, jsou packery schopny jej i šifrovat, obfuskovat nebo přidat k nim další ochrany (např. proti debugování nebo spuštění ve virtuálních počítačích). Bylo představeno 5 packerů s podrobným popisem jejich funkcionalit, se kterými se později pracuje v praktické části. Autoři malwaru však často používají modifikovaných verzí packerů, které jdou běžně sehnat na internetu, nebo si naprogramují vlastní, aby nebylo tak snadné zjistit, že je virus chráněn. Metod pro detekci komprimovaného souboru a použitého packeru existuje několik. Mezi nejpoužívanější patří detekce packeru pomocí signatur, kterou využívají detektory PEiD a ExeInfo PE, které byly taktéž představeny. Další metodou je detekce pomocí entropie souboru. U komprimovaných, šifrovaných či obfuskovaných souborů vzniká větší nahodilost v uspořádání jednotlivých bajtů, a tak je entropie vysoká a často se blíží až k maximální hodnotě 8. Dekomprese komprimovaných souborů může být provedena automaticky nebo manuálně. Pro automatickou dekompresi byly představeny nástroje zvané jako unpackery, které jsou určeny buď pro dekompresi konkrétního packeru (statické), nebo pro všechny packery (generické). Generické unpackery však doposud nejsou tak spolehlivé jako statické a největší problém mají s neznámými packery, které často využívají sofistikovanějších metod ochrany.
V praktické části byla implementována analyzační aplikace pro EXE soubory, která se snaží zjistit, zda je soubor komprimovaný či nikoliv, případně jakým packerem. Metody detekce využívané v této aplikaci byly inspirovány již existujícími studiemi o detekci komprimovaných souborů. Prvním způsobem byla tzv. podmínková klasifikace, kdy je z analyzovaného souboru extrahováno celkem 9 charakteristik podle jejichž hodnot je soubor klasifikován jako komprimovaný či nekomprimovaný. U této metody však nemusí nutně platit pravidlo, že čím méně charakteristik hlásí, že je soubor nekomprimovaný, tím menší je pravděpodobnost že komprimovaný doopravdy je. V druhé fázi bylo využito algoritmů strojového učení pro vytvoření modelů k binární a vícetřídní klasifikaci k detekci komprimovaných souborů a packerů. Úspěšnost modelů byla podle křížené validace vysoká, ovšem při aplikaci na jiných EXE souborech stažených z internetu byla značně nižší. Důvodem může být například nedostatečně velký, nebo naopak příliš velký datový set. Zabývat se podrobnou analýzou algoritmů strojového učení k detekci komprimovaných souborů a packeru však nebylo hlavní náplní této práce, a tak se jednalo pouze o pokus inspirovaný dosavadními studiemi. Dekomprese byla provedena automaticky pomocí

unpackerů představených v teoretické části a manuálně pomocí nástrojů OllyDbg a ImpRec. Manuální dekompresi lze pokládat za spolehlivější než automatickou, protože nám nabízí širší možnosti ve zkoumání kódu programu. V poslední části bylo provedeno pozorování změny entropie po aplikaci packeru, experiment s VirusTotal.com a experiment s vícenásobnou kompresí. Po aplikaci vícenásobné komprese na skutečný vzorek ransomware WannaCry bylo zjištěno, že více vrstev komprese, šifrování nebo obfuskace chrání škodlivý payload před antivirovými programy mnohem lépe, než pouze jedna vrstva.

Literatura

[1] SZOR, Peter. Počítačové viry: analýza útoku a obrana. Brno: Zoner Press, 2006. Encyklopedie Zoner Press. ISBN 80-86815-04-8.

[2] SIKORSKI, Michael a Andrew HONIG. Practical malware analysis: the hands-on guide to dissecting malicious software. San Francisco: No Starch Press, 2012. ISBN 1593272901.

[3] Malware analyst's cookbook and DVD: tools and techniques for fighting malicious code. Indianapolis: Wiley, 2011. ISBN 9780470613030.

[4] A Brief History of Malware Obfuscation *Cisco Blogs* [online]. 2010 [cit. 2018-02-18]. Dostupné z: `https://blogs.cisco.com/security/a_brief_history_of_malware_obfuscation_part_1_of_2`

[5] ESET TECHNOLOGY: The multilayered approach and its effectiveness [online]. [cit. 2018-12-2]. Dostupné z: `https://cdn1.esetstatic.com/ESET/US/docs/about/ESET-Technology-Whitepaper.pdf`

[6] Avast: Malware detection and blocking [online]. [cit. 2018-12-2]. Dostupné z: `https://www.avast.com/cs-cz/technology/malware-detection-and-blocking`

[7] Joshua Cannell: Obfuscation: Malware's best friend [online]. 2016 [cit. 2019-1-29]. Dostupné z: `https://blog.malwarebytes.com/threat-analysis/2013/03/obfuscation-malwares-best-friend/`

[8] Matt B: Malware Monday: Obfuscation [online]. 2016 [cit. 2019-1-29]. Dostupné z: `https://medium.com/@mbromileyDFIR/malware-monday-obfuscation-f65239146db0`

[9] Microsoft developer: PowerShell –EncodedCommand and Round-Trips [online]. 2014 [cit. 2019-1-29]. Dostupné z: `https://blogs.msdn.microsoft.com/timid/2014/03/26/powershell-encodedcommand-and-round-trips/`

[10] Encrypted and Oligomorphic Viruses [online]. 2015 [cit. 2019-1-30]. Dostupné z: `https://harrisonwl.github.io/assets/courses/malware/spring2017/slides/FinalWeeks/EncryptedOligomorphic.pdf`

[11] Wei Wang: Encrypted Viruses [online]. 2016 [cit. 2019-1-30]. Dostupné z: `http://www.cs.virginia.edu/~ww6r/CS4630/lectures/Encrypted_Virus.pdf`

[12] Malware Packers Use Tricks to Avoid Analysis, Detection [online]. 2017 [cit. 2019-1-30]. Dostupné z: `https://securingtomorrow.mcafee.com/business/malware-packers-use-tricks-avoid-analysis-detection/`

[13] Bat-Erdene, Munkhbayar & Kim, Taebeom & Park, Hyundo & Lee, Heejo : Packer Detection for Multi-Layer Executables Using Entropy Analysis. [online]. 2017 [cit. 2019-1-30]. Dostupné z: https://www.researchgate.net/publication/315320087

[14] Basic Packers: Easy As Pie | Trustwave | SpiderLabs | Trustwave [online]. 2013 [cit. 2019-1-30]. Dostupné z: https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/basic-packers-easy-as-pie/

[15] PE Format - Windows applications | Microsoft Docs [online]. 2019 [cit. 2019-1-30]. Dostupné z: https://docs.microsoft.com/cs-cz/windows/desktop/Debug/pe-format

[16] Wei Yan, Zheng Zhang, Nirwan Ansari - Revealing Packed Malware, 2008. [online] [cit. 2019-2-9]. Dostupné z: https://ieeexplore.ieee.org/document/4639028

[17] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, Wenke Lee - PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware, 2006. [online] [cit. 2019-2-9]. Dostupné z: https://nnt.es/polyunpack.pdf

[18] Lorenzo Martignoni, Mihai Christodorescu, Somesh Jha - OmniUnpack: Fast, Generic, and Safe Unpacking of Malware, 2008. [online] [cit. 2019-2-9]. Dostupné z: https://ieeexplore.ieee.org/document/4413009

[19] Github: crackinglandia/fuu [online]. 2018 [cit. 2019-4-9]. Dostupné z: https://github.com/crackinglandia/fuu

[20] ASM.Net - x86 Emulation - CodeProject [online]. 2018 [cit. 2019-4-12]. Dostupné z: https://www.codeproject.com/Articles/305152/ASM-Net-x-Emulation

[21] Write Your Own Unpacker - CodeProject [online]. 2018 [cit. 2019-4-12]. Dostupné z: https://www.codeproject.com/Articles/99873/Write-Your-Own-Unpacker

[22] Unpacking a UPX file manually with OllyDbg Hacking while you're asleep [online]. 2013 [cit. 2019-4-12]. Dostupné z: http://www.behindthefirewalls.com/2013/12/unpacking-upx-file-manually-with-ollydbg.html

[23] Downloads / Unpacking Tutorials - Tuts 4 You [online]. 2019 [cit. 2019-4-12]. Dostupné z: https://tuts4you.com/download/category/11//

[24] R. Lyda and J. Hamrock, "Using Entropy Analysis to Find Encrypted and Packed Malware,"in IEEE Security & Privacy, 2007. [online] [cit. 2019-2-9]. Dostupné z: https://ieeexplore.ieee.org/document/4140989

[25] UPX: the Ultimate Packer for eXecutables [online]. 2018 [cit. 2019-3-26]. Dostupné z: https://upx.github.io/

[26] ASpack Software - Application for compression, packing and protection of software [online]. 2018 [cit. 2019-3-26]. Dostupné z: http://www.aspack.com/

[27] PECompact – Windows (PE) Executable Compressor [online]. 2018 [cit. 2019-3-26]. Dostupné z: https://bitsum.com/portfolio/pecompact/

[28] PELock Software Protection & Software License Key System [online]. 2019 [cit. 2019-3-26]. Dostupné z: https://www.pelock.com/products/pelock

[29] Obsidium software protection system [online]. 2019 [cit. 2019-3-26]. Dostupné z: https://www.obsidium.de/show/home/en

[30] Softpedia: PEiD [online]. 2018 [cit. 2019-2-9]. Dostupné z: https://www.softpedia.com/get/Programming/Packers-Crypters-Protectors/PEiD-updated.shtml

[31] Aldeid: PEiD [online]. 2018 [cit. 2019-2-9]. Dostupné z: https://www.aldeid.com/wiki/PEiD

[32] Tuts 4 You: Download PEiD [online]. 2008 [cit. 2019-2-9]. Dostupné z: https://tuts4you.com/e107_plugins/download/download.php?view.398

[33] NTInfo: Deteci It Easy [online]. 2018 [cit. 2019-2-9]. Dostupné z: http://ntinfo.biz/index.html

[34] Github: horsicq/Detect-It-Easy [online]. 2018 [cit. 2019-2-9]. Dostupné z: https://github.com/horsicq/Detect-It-Easy

[35] Collberg, C., Thomborson, C., Low, D.: A Taxonomy of Obfuscating Transformations [online]. 1998 [cit. 2018-10-30]. Dostupné z: https://researchspace.auckland.ac.nz/bitstream/handle/2292/3491/TR148.pdf

[36] Collberg, C., Thomborson, C., Low, D.: Breaking Abstractions and Unstructuring Data Structures [online]. 1997 [cit. 2018-10-30]. Dostupné z: https://ieeexplore.ieee.org/document/674154

[37] Cappaert J.: Code Obfuscation Techniques for Software Protection [online]. 2012 [cit. 2018-10-30]. Dostupné z: https://www.esat.kuleuven.be/cosic/publications/thesis-199.pdf

[38] Paladion: Code Obfuscation [online]. 2005 [cit. 2018-11-24]. Dostupné z: https://www.paladion.net/blogs/code-obfuscation

[39] Paladion: Code Obfuscation - Part 2: Obfuscating Data Structures [online]. 2005 [cit. 2018-11-24]. Dostupné z: https://www.paladion.net/blogs/code-obfuscation-part-2-obfuscating-data-structures

[40] Paladion: Code Obfuscation Part 3 - Hiding Control Flows [online]. 2005 [cit. 2018-11-24]. Dostupné z: https://www.paladion.net/blogs/code-obfuscation-part-3-hiding-control-flows

[41] Top 6 Advanced Obfuscation Techniques Hiding Malware on Your Device [online]. 2017 [cit. 2018-11-24]. Dostupné z: https://sensorstechforum.com/advanced-obfuscation-techniques-malware/

[42] Github: secana/PeNet [online]. 2019 [cit. 2019-4-4]. Dostupné z: https://github.com/secana/PeNet

[43] ML.NET | Machine Learning made for .NET [online]. 2019 [cit. 2019-4-4]. Dostupné z: https://dotnet.microsoft.com/apps/machinelearning-ai/ml-dotnet

[44] Strings - Windows Sysinternals | Microsoft Docs [online]. 2019 [cit. 2019-4-4]. Dostupné z: https://docs.microsoft.com/en-us/sysinternals/downloads/strings

[45] PE Format - Windows applications | Microsoft Docs [online]. 2019 [cit. 2019-4-4]. Dostupné z: https://docs.microsoft.com/en-us/windows/desktop/debug/pe-format

[46] Roberto Perdisci, Andrea Lanzi, Wenke Lee: Classification of Packed Executables for Accurate Computer Virus Detection [online]. 2008 [cit. 2018-10-30]. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.188.2114&rep=rep1&type=pdf

[47] Github: fabrimagic72/malware-samples [online]. 2017 [cit. 2019-4-20]. Dostupné z: https://github.com/fabrimagic72/malware-samples/tree/master/Ransomware/Wannacry

A Seznam názvů standartních PE sekcí

Seznam názvů standartních PE sekcí vyextrahovaných z celkem 725 EXE souborů z adresáře `C:\Windows` operačního systému Windows 10 Home:

```
.text
.rdata
.data
.rsrc
.reloc
.idata
.didat
.imrsiv
RT_CODE
RT_BSS
RT_DATA
RT_CONST
PAGE
consent
.tls
PAGER32C
.wpp_sf
KVASCODE
POOLCODE
ALMOSTRO
CFGRO
CACHEALI
PROTDATA
PAGELK
PAGEKD
PAGEVRFY
PAGEHDLS
PAGEBGFX
.edata
PAGEDATA
PAGEKDD
PAGEVRFD
INIT
INITDATA
.tPolicy
```